FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Cross-platform data-driven applications with React Native and GraphQL: Principles and practices

Gonçalo Maria Nunes Andrade Lobo dos Santos

DISSERTATION REPORT

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: António Miguel Pontes Pimenta Monteiro

July 22, 2017

Cross-platform data-driven applications with React Native and GraphQL: Principles and practices

Gonçalo Maria Nunes Andrade Lobo dos Santos

Mestrado Integrado em Engenharia Informática e Computação

July 22, 2017

Abstract

The current state of mobile development often demands that a certain compliance with different operating systems and smartphones is met. The most typical approach to solve this problem consists in a singular native development for each platform, mainly one for Android and another for iOS (both current market leaders in operating systems for mobile devices). This process turns out to be awfully time-consuming and more expensive as it requires developers to have expertise in two different sets of knowledge. In order to answer this problem, several cross-platform approaches were pursued but few are able to provide a fluid and native feeling as result applications.

The main objective of this thesis dwells in a development study performed on one of the newest cross-platform approaches, React Native, which concurrently allows the creation of iOS and Android applications by compiling code written in JavaScript and by using the same UI components from each system (providing a more native feeling).

Another key objective for this thesis doesn't rely on the front-end aspect of application development but on the data fetching and manipulation. Most web services are built around REST and its strict specifications in order to enable a proper interoperability between systems. With that in concern, Facebook announces GraphQL, a query language suitable for APIs, allowing developers to define their data using a fully-fledged type system, forming a schema that is self-documenting and giving clients full control over the data they request. Practices and patterns for application development while using previously mentioned tools were extracted and are presented alongside a thorough analysis of their inherent principles.

Resumo

O estado atual do desenvolvimento de software para telemóveis requer frequentemente que exista conformidade entre diferentes sistemas operativos e dispositivos. Para alcançar esse objetivo, tipicamente opta-se por levar a cabo o desenvolvimento nativo de diferentes aplicações em que cada uma é orientada para uma única plataforma.

Este processo de desenvolvimento revela-se demorado e caro pois exige que os programadores de uma empresa tenham conhecimento mais alargado e que sejam polivalentes em várias linguagens. Ao longo da última década foram emergindo técnicas de desenvolvimento multi-plataforma mas poucas são capazes de atingir a fluidez e a sensação de utilização que as aplicações nativas trazem.

Esta tese tem como objetivo levar a cabo um estudo focado numa das mais recentes abordagens de desenvolvimento multi-plataforma, React Native, que permite a criação em simultâneo de aplicações iOS e Android compilando código escrito em JavaScript e fazendo uso dos mesmos componentes nativos de cada sistema (proporcionando assim uma experiência mais nativa aos utilizadores).

Outro objetivo desta dissertação desvia-se do desenvolvimento de aplicações para o lado do cliente e dá ênfase em aspetos relacionados com a busca e manipulação de dados. A maioria dos serviços web são construídos utilizando REST. Esta abstração está normalmente associada a especificações rigorosas que permitem que haja interoperabilidade adequada entre sistemas. Numa tentativa de simplificar as normas atuais surge GraphQL, uma linguagem para interrogar APIs, que permite que os programadores definam os dados e os tipos de objetos de um sistema permitindo ao mesmo tempo que o cliente tenha a liberdade de solicitar exatamente os dados que necessita.

Boas práticas e padrões de desenvolvimento de aplicações usando React Native e GraphQL foram extraídos e são apresentados juntamente com uma análise cuidada dos seus princípios fundamentais como tecnologias.

Contents

List of Figures

LIST OF FIGURES

List of Tables

LIST OF TABLES

Abbreviations

API	Application Programming Interface
ESn	ECMAScript version n
DOM	Document Object Model
LoC	Lines of Code
IDC	International Data Corporation
IDE	Integrated Development Environment
npm	Node Package Manager
OS	Operating System
RN	React Native
SVG	Scalable Vector Graphics
UI	User Interface
UX	User Experience

Chapter 1

Introduction

Eminent cross-platform tools destined for mobile development have been emerging throughout the years but few are able to provide a fluid and native feeling as result applications. This thesis will focus on app development using React Native and GraphQL (both publicly announced and released in 2015). Several methodologies and techniques will be explored in order to draw out key practices that will justify the integration of such tools in the processes and organization of a development team. This thesis also has the objective of enabling more businesses to develop apps for all platforms, avoiding native programming.

1.1 Context

With the turn of the century there was a noticeable growth in the smartphones market. This growth was concurrent with the release of both the iPhone (2007) and Android (2008), which drew a new kind of audience, revolutionizing the mobile market. Before that, smartphones were mostly targeted for enterprise environments instead of casual users.[1]

In order to match this growth there was an evident urge for developers to build extra features that lacked on the devices' operative system. Later came application stores such as App Store for iOS and Google Play for Android. Using these platforms developers could now deploy applications to be used by anyone for free or for an established monetary cost (similar to what happens with computer programs).

Since there are several operating systems, knowing that each system can be ran on a wide range of different devices, it becomes evident that there must be a concern while developing an application if one wants it to have a greater reach. That being so, it is generally mandatory for applications to be developed for more than one platform which forces companies to hire developers with a broader set of skills.

Thereafter, cross-platform development appears in a way where a single implementation can work on distinct platforms. This approach grants benefits such as a higher rate of code reuse,

due to the ability to share code between platforms, reducing development costs and increasing productivity.

1.2 Motivation

The topic for this thesis has been proposed and was carried out in cooperation with Glazed Solutions, a software company based in Porto, Portugal. Glazed Solutions offers an end-to-end product development to their costumers who often demand the development of web services and mobile applications. This scenario typically results in Glazed Solutions having to develop two single mobile applications, one for iOS and another for Android. If the company switched to development using cross-platform technologies like React Native the developing time would shorten as they could deploy the applications and other services using similar code.

1.3 Goals

The main goal of this thesis is to evaluate React Native's framework and analyze how well the developed applications perform. Furthermore, an evaluation on GraphQL will be carried on. These evaluations will hopefully allow the extraction of good practices for development of applications using the mentioned tools as well as better insight on how these affect the processes and organization of a development team.

Chapter 2

State of The Art of Mobile Development

This chapter presents an extended theoretical background on mobile development allied to a brief introduction on the technologies used during this thesis.

Broader concepts related to the development of apps for smartphones are firstly introduced focusing mainly on OS and application structure both for Android and iOS. Furthermore, different cross-platform approaches are reviewed as well as the React Native approach which will be the target to several evaluation techniques on its performance and also on its look and feel.

Afterwards, theoretical work on web services interfaces is presented. These usually consist of collections of operations intended to be used by clients over the internet. The chapter goes on exploring serialization and data formats, relevant concepts for addressing the subject of data fetching. An overview on REST and the technologies intended for evaluation during this thesis, namely GraphQL, follows.

Finally, the chapter concludes presenting a couple of design patterns for app development and evaluation methods using metrics such as the cost of said development and quality of the processes.

2.1 Smartphone applications

Smartphones are considered high-end cellphones more powerful than the *feature phone* and have larger, high-resolution screens and more device capabilities [1]. In order to function as intended, smartphones have to run an OS that allows the installation of advanced, third-party programs (apps) which can be typically obtained by accessing the OS's manufacturer marketplace [2]. According to the International Data Corporation (Table 2.1), as of November 2016 Android dominates the smartphone market with a share of 86.8% of unit shipments followed by iOS (with 12.5%) [3].

Table 2.1: Worldwide Smartphone OS Market Share

Period	Android	iOS	Windows Phone	Others
2015Q4	*79.6%*	*18.7%*	*1.2%*	*0.5%*
2016Q1	*83.5%*	*15.4%*	*0.8%*	*0.4%*
2016Q2	*87.6%*	*11.7%*	*0.4%*	*0.3%*
2016Q3	*86.8%*	*12.5%*	*0.3%*	*0.4%*

2.1.1 Android

Android Inc. was funded in 2003 and acquired two ears later by Google, which now leads the Android Open Source Project which is responsible for maintaining the software. As for the hardware, it is supported by the Open Handset Alleance (OHA) which is a conglomeration of many handset manufacturers. Applications destined to run on Android OS are written in Java [4].

2.1.1.1 Architecture

The Android operating system architecture can be segmented into four different layers (see figure 5):

- **Linux Kernel**
  Linux is at the bottom layer providing basic system functionality like process management, memory management, device management. The kernel also handles the networking and has a vast array of device drivers [5].

- **Libraries and Android Runtime**
  Above the Linux kernel are Android's native libraries which enable the device to handle different types of data. Written in C or C++, these libraries are called through a Java interface. Among others, there are libraries to record and play audio/video, SSL libraries responsible for internet security and the web browser engine WebKit [5]. Located at the same level there's the Android Runtime which is an application runtime environment that performs the translation of the application's bytecode into native instructions intended to be executed by the device's runtime environment [6].

- **Application Framework**
  This layer provides several high-level services to applications in the form of Java classes, allowing developers to make use of them in their applications [5].

- **Application**
  On the top there's the Application layer where some applications come pre-installed with every device (such as SMS client app and contact manager). Developers can write their own applications which will also be installed here [5].

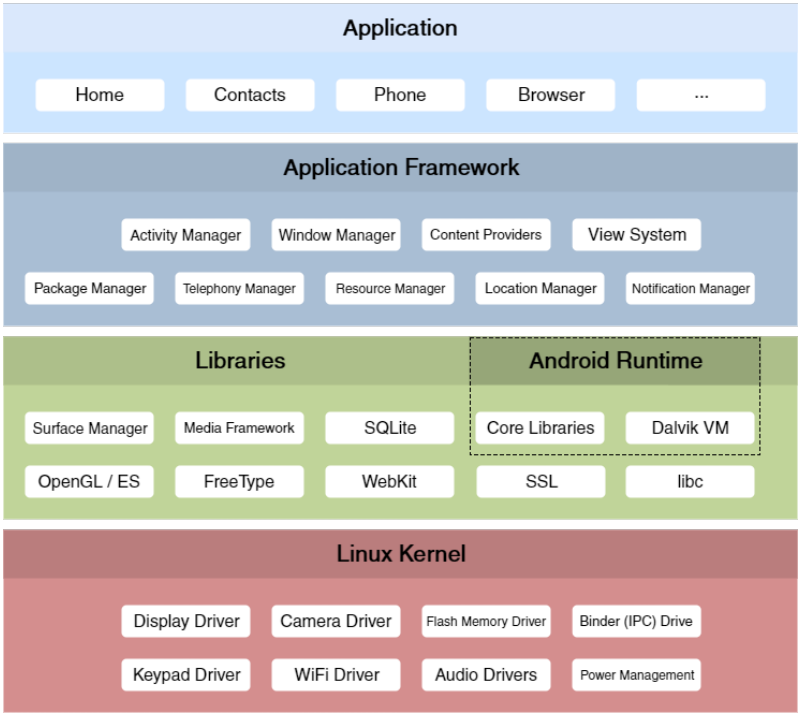


Figure 2.1: Android Arquitecture

2.1.1.2 Development

There are integrated development environments that facilitate the creation of mobile applications such as Eclipse and Android Studio, the official IDE for Android development [7]. When creating a new project the Android SDK aids the developer structuring their app and takes advantage of the Java Android Library which contains all the packages, application framework and class libraries that are required during the development. There are specific Android classes such as the Activity class, but the general syntax is the same as the original Java. After project creation it is possible to compile the Java code which will result in an application consisting of an Android Package (APK) that can be installed and ran on a device. Every Android version comes with a different Android Virtual Device (AVD) that can be launched in the IDE which will allow an emulation of an Android device for possible testing and debugging purposes. UI components in Android are declared as Views and are created and presented by Activities. The styling or layout is provided by an XML file [8].

2.1.2 iOS

iOS is an OS developed by Apple and its use is exclusively set for devices produced by the same company. It was publicly announced in 2007 in parallel with the iPhone. The system has been updated throughout the years and, as of now, stands on version 10 [4] [9].

2.1.2.1 Architecture

iOS applications do not communicate directly to the fundamental hardware components. So, to perform this communication, iOS presents a layered architecture with a set of well-defined interfaces.

This architecture approach is structured into four distinct layers each one serving a different purpose (see figure 2.2):

- **Core OS**
  iOS's bottom layer aggregates the low-level features which are encapsulated and used by other layers. Typically, its functionalities are not directly manipulated by the developers but represent a liability for supporting layers at a higher level [10].

- **Core Services**
  This layer is responsible for providing iOS applications with core system services such as location and networking. Among these services there's the JavaScript Core framework that bears the purpose of executing JavaScript code and parsing JSON data [10].

- **Media**
  As the name might indicate, this is the layer that allows developers to implement audio or graphic features on their applications. The Media layer provides access to APIs that are capable of dealing with numerous aspects regarding media [10].

- **Cocoa Touch**
  The Cocoa Touch layer supplies the developer with the frameworks that define the application's appearance. Additionally, it provides the app's basic infrastructure and access to high-level system services [10].

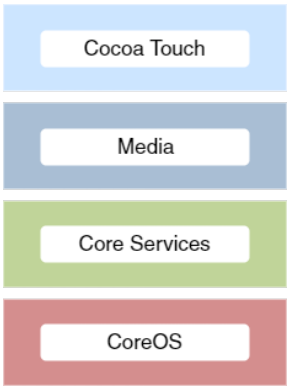


Figure 2.2: iOS Arquitecture

2.1.2.2 Development

In order to develop applications for iOS there is a set of mandatory requirements that need to be fulfilled. These requirements are related to the development environment which demands usage of specific hardware and software, namely a Mac computer running macOS 10.11.5 or later, and Apple's official IDE, Xcode. Another requirement consists in having the iOS SDK which extends Xcode's functionalities. This SDK comes with Xcode's installation along with a simulator which can be used for testing and debugging purposes. As for coding itself, iOS development languages are Objective-C, Swift or a combination of both if needed [11].

2.1.3 Cross-platform approaches

Cross-platform development approaches emerged with the purpose of allowing developers to implement their apps at once for more than one platform. It is typically desired that these approaches offer a certain generality but this often implies having a lowest common denominator feature set in order to grant provision to several devices [12].

There are tools for cross-platform development of applications which rely on different architectural approaches. A description of several approaches is presented next.

2.1.3.1 Web approach

The first obvious approach consists in building an application as a mobile web app destined to be executed by the device's web browser. Therefore, web applications will be platform independent and based on standard internet technologies such as HTML, CSS and JavaScript. The client side of the application deals with the user interface and data validation logic while other critical operations take place on the server side.

The implemented applications can be accessed using a URL in the web browser without requiring previous installation or subsequent updates as the data and the applications are hosted on the server. The time and download cost of rendering web pages, the fact that internet connection is usually a requirement and the limited access to the device's hardware capabilities are some of the disadvantages of the web approach [13]. Sencha Touch is an example of a user interface JavaScript web framework that follows this approach [14].

2.1.3.2 Hybrid approach

This approach tries to combine features from both web and native apps. These applications don't require previous knowledge of the target platform and are built using the same above mentioned web technologies. Instead of running in a browser, hybrid apps get executed physically on the device (require installation) inside a native container. It is possible to access some of the device's hardware capabilities through specialized APIs.

A disadvantage of these apps is that they are susceptible to platform specific behavior of JavaScript interpretation as well as threading model incompatibilities. Their inferiority when compared to native apps rely on the fact that they cannot completely achieve a native look and feel and tend to require more processing time [13].

Apache Cordova (formerly PhoneGap) is a popular hybrid mobile application development framework [15].

2.1.3.3 Cross-compilation approach

Cross compiler generated applications are compiled similarly to native apps. Developers utilize a framework that provides a platform independent API to build the application. Cross-platform tools such as Xamarin [16] transform the produced code into native [13]. At a look and feel level cross--compilation doesn't show disadvantages in comparison with native programming. However, it is complex to implement and achieve consistency between different devices and operating systems [17].

2.1.3.4 Interpreted approach

This approach relies on an interpreter that interprets the code on runtime across different platforms. Native code is automatically generated and implemented onto the UI, meaning that end users will always interact with platform specific native UI components while the logic is implemented separately. The native API can be accessed through an abstraction layer that exposes system innate features to the developer allowing device's hardware capabilities to be used. This approach relies heavily on the interpreter's framework which can limit the available features for development [12].

Adobe AIR is an example of a tool that adopts this approach [18].

2.2 Web services

Web services supply servers with an interface that can be used to interact with client applications using standard web protocols. Accordingly, this interface consists of a collection of operations, intended to be invoked over the web, triggering the execution of a program on the server which can then return a corresponding result or initiate other operations [19].

2.2.1 REST

REST (Representational State Transfer) is a software architecture style targeted for systems in which the integrating components are not on the same physical machine and, for that reason, need to communicate through a network. This systems are called distributed information systems [20].

2.2.1.1 Principles

REST relies on two key aspects which are resources and representations. A resource can be stored on a computer and represented as a stream of bits, it can be anything relevant enough to be referenced as a thing itself, for instance, it may be a person, an apple or even an abstract concept like hunger [21]. Representations are used to transfer resources' state between different components and may assume binary or textual representation (as referred on 2.2.2).

2.2.1.2 Constraints

REST's constraints are high-level and do not specify implementation-specific protocols or file formats because of its architectural nature. This style is formally characterized by a set of six constraints [20]:

- **Client-Server**
  Client and server functionalities have to be separated. This measure promotes portability on the client-side since there's less concern with data storage which, along with the business logic, can be located on the server-side. Consequently this improves scalability as the server components are kept simple and separated from the UI [20].

- **Stateless**
  This constraint indicates that it is not allowed to keep stateful session data on the server-side, instead it's the client-side's role to include all the required information to process a request. On the one hand scalability and simplicity of the servers is ensured as it is not necessary to store any state between requests. On the other hand this constraint can decrease communication performance between components as the clients have to repeatedly send more data to ensure consistency [20].

- **Cache**
  In order to improve efficiency, data within responses must be specified as cacheable or not. This way a client can assure if retrieved data can be stored for later use instead of repeatedly inquiring the server. Nonetheless, there could be a problem regarding reliability as cached data could become obsolete and differ from the server's [20].

- **Uniform interface**
  Developed interfaces should promote a uniform and simple architecture. This allows interactions between components to be performed in a more standardized way but could compromise optimal settings for data transfer [20]. To improve this uniformity, reducing its downsides, there are four additional constraints:

  - **Identification of resources**
    Each available resource must be identifiable in order to be recognizable during interactions [20]. A common practice of this principle is implemented in HTTP where different resources have distinct URLs [19].

  - **Manipulation of resources through representations**
    Resources are added, removed or updated by sending representations with embedded control data between clients and a server [20]. HTTP verbs (GET, POST, PUT, DELETE) are representations of control data, given that they supply information on how a certain resource is to be manipulated [19].
  - **Self-descriptive messages**
    Besides data, exchanged messages between components also contain information (metadata) describing how said message is to be processed. [20].
  - **Hypermedia as the engine of an application state**
    Clients can trigger a state change in the system only through state transitions provided by the server [20].

- **Layered system**
  Components should be structured in layers and only have access to other components within the layer they're interacting with. This measure helps reduce the system's complexity [20]. If needed, there's the possibility of creating intermediary components (such as proxies or firewalls) that function between server and client and can process or transform sensitive data [19]. A downside of this layered structure is that it can increase latency while messages go through several different layers.

- **Code-on-demand**
  This constraint enables extension of the clients' functionalities in a way that allows them to execute code received from a server. This last constraint is considered as optional on a REST architecture [20] but it has been implemented on web browsers, for example, where they download and execute JavaScript code provided from a server.

Despite not settling for a specific protocol regarding to the exchanged messages, most REST API implementations follow HTTP standards [22].

2.2.2 Serialization and data formats

When there is the need of transferring an object between two sub-systems over the web it's necessary to convert into a suitable format. This process is called serialization and its product is called external data representation, which can be either binary or textual [19]. In binary representation the data is serialized into an unreadable format (such as Google's Protocol Buffers [23] and Java Object Serialization [24]) or into textual representation formats (like XML [25] and JSON [26]). Each one of these approaches have their own trade-offs and will affect the performance of data transmission in different ways. Although the size of serialized data using textual representation tends to be larger, the unserialization may be quicker [27].

2.3 JavaScript

JavaScript is one of the most common programming languages in web and mobile development [28] [29]. It is a dynamic, high-level, untyped interpreted language which was initially released to create simple interactions on web pages. Nowadays it can be found and used pretty much anywhere, from microcontrollers to mobile apps. Its core language features are defined by a standard denominated as ECMA-262 [30].

2.3.1 ECMAScript

ECMAScript is a scripting language specification standardized by ECMA international in ECMA-262. It is currently in its seventh edition of standardization which is also known as ECMAScript 2016. Dynamic web applications growth in 2007 increased JavaScript's popularity and, because JavaScript had not had any update since its third version (1999), a large revision took place. The new proposed specification was massive in scope and presented changes in several features of the language. Although it was time for some adjustments many committee members felt that ECMAScript was trying to accomplish too much too fast. Those ambitious changes were only implemented later in 2015 with ECMA6 featuring modifications like new objects, patterns, methods and syntax changes. The fact that most browsers do not fully support JS newest specifications represents an obstacle to its standardization. A JS transpiler, like Babel, is almost mandatory if developers want to partake in JavaScript's newest specifications. These transpiling tools allow developers to have well structured and maintainable code without having to worry about browser support [30].

2.3.2 NPM

NPM is the default package manager for several JavaScript runtime environments. It maintains track of all the packages and their versions, allowing developers to update and/or remove them. Each project has its external dependencies stored inside a file called *package.json*. Using npm, client developers are able to install packages that get downloaded from a dedicated registry [31].

Chapter 3

Technical Principles

3.1 React Native

React is a JavaScript library used for building user interfaces that was created to enhance applications' page rendering and data-fetching performance. In the earlier days of web development, pages would reload whenever a change on the display was needed or whenever a user performed an action. With the advent of Single Page Applications a change on this course of action was eminent as it was expensive to reload a complete page. React tries to solve this problem from the View layer introducing a new mindset to front-end development.

React presents its views as abstractions and splits them into *Components*. Each component is responsible for describing the view itself and the logic accountable for handling its display.

Another of React's key concepts is the virtual DOM. Instead of rendering directly to the actual DOM, React will render a virtual DOM which is used to calculate the differences between both DOMs. This way it's possible to compute the minimum number of DOM operations necessary to achieve a new state.

React Native (RN) is a JavaScript framework for creating real, natively rendering mobile applications. It is based on the previously mentioned library, React, and follows most of its key concepts. Using RN, developers are able to share code between platforms simplifying concurrent development for both Android and iOS, building mobile applications which genuinely look and feel native [32].

RN applications are written using JavaScript and JSX, a markup language similar to XML [33] [25]. After being written, RN invokes the native rendering APIs in Java (Android) and/or in Objective-C (iOS) producing real mobile UI components instead of webviews as implemented on other JavaScript based cross-platform approaches (section 2.1.3). RN also provides access to platforms' APIs allowing the access to the devices' hardware capabilities [32].

3.1.1 Core Concepts

RN makes use of the same core concepts considered to be the bedrocks of React development. A thorough understanding of these concepts allows developers to optimize the use of this technology.

3.1.1.1 Virtual DOM

In most web applications, in order to build interactive UIs, developers edit directly the DOM, which is considered an expensive procedure as it severely impacts performance. React for web, upon its release, introduced an innovative factor which reduced the amount of times that the editing had to occur. Instead of fully rendering a page, React would first compute and then render the necessary changes by using a stored memory of the DOM [32].

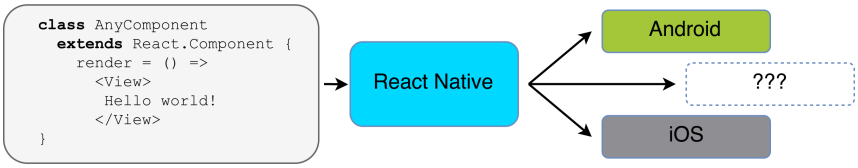


Figure 3.1: Component rendering using React Native

React Native also builds and internally maintains a representation of the rendered UI. This representation helps React avoid creating/accessing DOM nodes beyond necessity. This abstraction is referred to as virtual DOM and it works similarly on RN. Instead of rendering to the browser's DOM, React Native invokes each platform's API to render Android and/or iOS components (see figure 3.1). Components in React return JSX markup from their *render()* method, with the description of their appearance, which are then translated to suit the host platform. [32].

3.1.1.2 One-way Data Flow

React was built having in concern the challenges of MVC frameworks. Most MVC views encompass a heavy UI and it is common practice to have a cycle that forces the app to re-render as the data mutates. The main reason behind this method relies on the fact that most MVC frameworks follow a two way data binding (see figure 3.2).

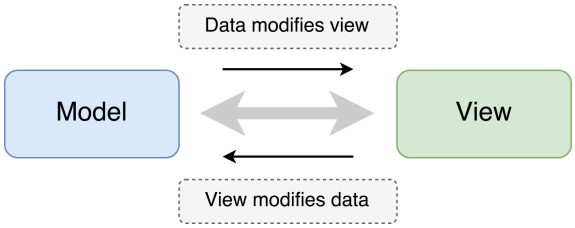


Figure 3.2: Two-way data binding.

Applications destined to be used in the real-world often have multiple views representing data in one or more models. The more complex they get, the more models and views are required. The two-way data binding can easily become catastrophic making it hard to predict the effect of changes on data. A view could update a model, which in turn could update a view, and so on. If the chain effects are not properly tracked it is easy to end up in an infinite event loop (see figure 3.3).

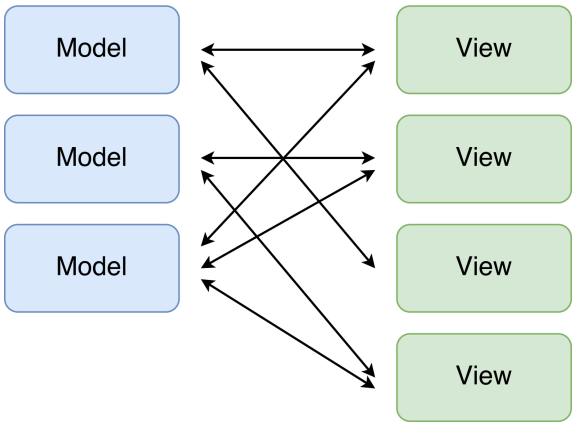


Figure 3.3: Complex MVC application.

This unpredictability makes introducing new developers to a code base a hard task. Even new developers should be able to tell with some degree of reliability which impact one change might cause.

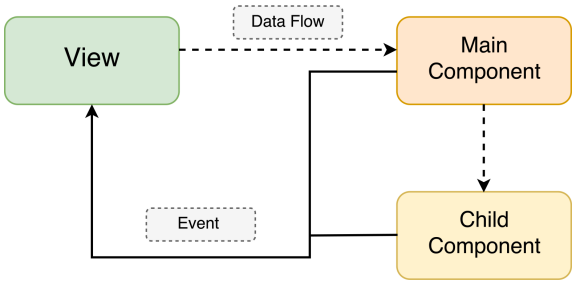


Figure 3.4: One-way Data Flow

React's one-way data flow (see figure 3.4) is based on a design principle (Separation of Concerns). This principle states that an application should be divided into distinct sections where each one serves a single purpose. By doing this segmentation, development becomes simplified and improves a system's capability to scale and be maintained [34].

3.1.1.3 Declarative Components

React Native applications are built using composable, modular components that allow the splitting of the UI into independent and reusable pieces. The general idea is to build components that manage their own state and then compose them to create complex UIs. Conceptually, components behave like JavaScript functions. They accept inputs called *props* and return React elements describing what should appear on the screen. React favors composition over inheritance. Components are disposed hierarchically, starting with a root and being able to branch into other components (children).

```
import React, { Component } from 'react';

class Message extends Component {
  render() {
    return (
      <Text>
        Greetings {this.props.name}!
      </Text>
    )
  }
}
export default Message
```

Listing 3.1: Simple component.

Components, in their most simple form, only require that a *render()* method is defined. Listing 3.1 is an example of a simple component which expects a *prop* declared as *name* to be sent by a higher-level component in order to display a greeting message as unformatted text. It is also possible to set default values for *props* and define component's *PropTypes*, that formally inform about the value types that can be expected for certain *props*. Contrarily, components can be completely static and unable to receive external inputs, always rendering exactly the same way [35].

Developers can declare already existing components and build them like blocks to create more complex components (see listing 3.2).

```
import React, { Component } from 'react';
import Message from './Message'

class MutiMessage extends Component {
  render() {
    return (
      <Message name="Goncalo"/>
      <Message name="Catarina"/>
    )
  }
}

export default MassGreeting
```

Listing 3.2: Component composition.

A component can perform external, read-only, operations that are passed as *props*. It can also keep track of its internal state which is convenient to manage visual display options (for instance, should a button be visually enabled or disabled).

To manage internal state, components must define initial values of internal state variables by declaring them in the constructor. These variables allow developers to use the internal state to perform conditional rendering. The attributed value for the inital state variables can be internally

changed by the component, by using the *setState()* method. Any changes on the internal state will trigger the execution of the *render()* method forcing an update on the UI. [36]

React components go through several steps before and after rendering. Developers are able to parse through these steps in order to perform tasks or check for specific conditions.

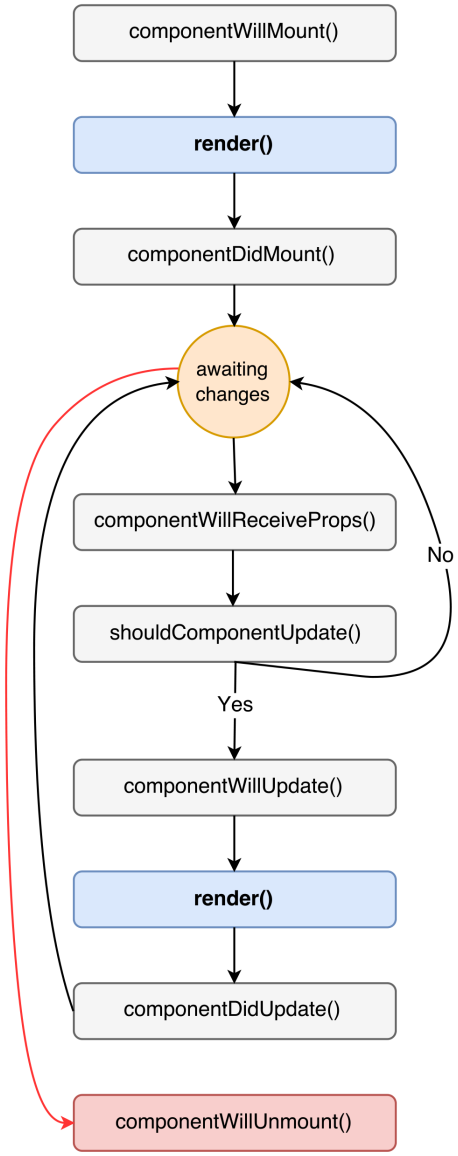


Figure 3.5: Components' lifecycle methods.

These steps are referenced to as the component lifecycle and are presented on figure 3.5. As it can be noticed, the *componentWillMount()* method will always trigger before the *render()* method and will be followed by *componentDidMount()*. Besides the methods that are reponsible for managing the update cycle (*componentWillReceiveProps()*, *shouldComponentUpdate()*, *componentWillUpdate()* and *componentWillUpdate()*) there is also a method that is called after a component is removed, which can be used to clear up anything that the component created while

existed [37].

The steps were named in a self-explanatory way and are only executed if the particular component has a method with the method's name defined.

3.1.2 Architecture

In React Native, layout calculations and JavaScript execution are kept separate from the main thread which focuses on performing native tasks. These tasks are responsible for dealing with aspects such as the rendering of native views or the handling of touch events.

As previously stated, components help represent the application state. An update on the interface is prompted when there is a change to a state or prop of a component. If that occurs, RN will trigger its update lifecycle asynchronously by assembling the updates necessary and passing them across a bridge that establishes communication with the native side of the devices. Although it is important to have an insight of how things work in RN, most technical aspects can be ignored during development allowing the developers to target their effort into JavaScript coding.

The React Native Packager, responsible for bundling all the constituents of the applications, runs Babel which is a tool that allows transpiling from ES6 (and newer versions) JavaScript into ES5 (see section 2.3.1). This way, older devices that support former JavaScript runtimes can also leverage from fresh JavaScript features [38].

3.2 GraphQL

GraphQL is a query language designed to provide an intuitive and flexible syntax for describing data requirements and interactions of client-server applications. It aims to provide a unified interface by allowing application servers to take their capabilities and map them to a uniform language and type system [39]. Besides functioning as a language, GraphQL is also a runtime, an execution layer that servers can use to process requests [40]. GraphQL relies on several key concepts:

- **Hierarchical**
  GraphQL queries are shaped similarly to the data they return. Thus developers benefit from having a *natural way* for describing data requirements [39].

- **Product-centric**
  GraphQL enables building the necessary language and runtime in accordance to the requirements of views and to the front-end engineers' way of thinking [39].

- **Strong-typing**
  GraphQL servers define an application-specific type system demanding interrelated queries to be executed within its context. Given a query, it must be possible to ensure its validation before execution. Servers can also give information relative to the shape and nature of the responses [39].

- **Client-specific queries**
  Each GraphQL server, by defining its type system, publishes a set of capabilities that its clients are allowed to handle. Using that information, clients are responsible for encoding their queries which can be specified at field-level granularity. It is not the server's role to determine which data is to be sent, it simply returns exactly what the client asked for [39].

- **Introspective**
  Clients are able to query the type system by using GraphQL's syntax [39]. This represents a powerful feature for statically typed languages, like Java (2.1.1.2), Swift or Objective-C (2.1.2.2), as it makes error-prone code to parse untyped JSON objects (into strongly-typed ones) no longer necessary.

With GraphQL is possible to engaged in data-driven programming as the client describes the data to be matched and the processing required.

3.2.1 Type system

GraphQL's type system is used for query validation. Moreover, it describes the input types of query variables to ascertain that the values provided at runtime are valid. The type system also expresses the capabilities of a server by defining a schema. It supports features such as scalars, interfaces, unions, enumerations, lists and more. Listing 3.3 presents an example of a schema

where types share a common interface and, therefore, inherit its attribute fields. The Vehicle interface defines two object types, Car and Bicycle. It is possible to retrieve information about the owner of the vehicles as Person is also defined as a type. Since every field is nullable by default an exclamation mark is needed to prevent fields from being null [41].

```
type Person {
  id: ID!
  name: String!
  birth: Date!
  nationality: String
}

interface Vehicle {
  owner: Person!
  color: String
}
```

```
type Car implements Vehicle {
  owner: Person!
  color: String!
  model: String!
  plateNo: String!
}

type Bicycle implements Vehicle {
  owner: Person!
  color: String
}
```

Listing 3.3: Example of GraphQL's type definitions.

Declaring the Query type is mandatory because it functions as an entry point for validating and executing queries. An example on the implementation of a Query type is shown on listing 3.4 where three different operations are defined:

- ***person***, as the name indicates, returns a Person object type by passing an ID number;
- ***vehicle*** returns a Vehicle object type by passing information on the owner;
- ***car*** returns a Car object type that matches a given plate number.

```
type Query {
  person(id: ID!): Person
  vehicle(owner: Person!): Vehicle
  car(plateNo: String!): Car
}
```

Listing 3.4: Example of a GraphQL's Query type.

3.2.2 Operations

Essentially, there are two kinds of operations when dealing with a GraphQL server. Operations can either serve the purpose of querying or mutating. In other words, queries represent read operations and mutations represent write operations. Regardless of which, these operations are carried out

sending a string that the GraphQL service will resolve. The typical service's response format, when dealing with mobile applications, is JSON [40].

With GraphQL it is the request that determines the JSON response's structure so one can say that both communication routes are heavily related (see listings 3.5 and 3.6).

```
query person(id: 13) {
  name
  country
}
```

Listing 3.5: Example of a GraphQL query.

```
{
  "data": {
    "person": {
      "name":"Catarina Marques",
      "country":"Portugal"
    }
  }
}
```

Listing 3.6: Example of a JSON response.

```
{
  english:
    person(country: "England") {
      ...commonFields
    }
  portuguese:
    person(country: "Portugal"){
      ...commonFields
    }
}

fragment commonFields on Person {
  id
  name
}
```

Listing 3.7: Example of a GraphQL fragment query.

```
{
  "data": {
    "english": [
      {
        "id": 1,
        "name": "John Doe"
      },
      {
        "id": 2,
        "name": "Jane Doe"
      }
    ],
    "portuguese": [
      {
        "id": 7,
        "name": "Goncalo Lobo"
      },
      {
        "id": 13,
        "name": "Catarina Marques"
      }
    ]
  }
}
```

Listing 3.8: Example of a JSON response to a fragment query.

3.2.3 Fragments

GraphQL excels in cases where applications need to combine several UI components with different sets of data in the same page. To prevent having multiple and complex queries, GraphQL introduces a new feature called fragments.

Using fragments the developers can construct sets of fields and then include them on their queries. This way it is possible to compose queries while avoiding code repetition, as the query fields are only declared once, and ensuring there's only one initial data fetch [40].

3.2.4 Apollo GraphQL Server

Apollo is a flexible, production-ready GraphQL client that allows integration with React Native apps. Essentially, the Apollo Client allows integration with any GraphQL server. It enables developers to be flexible regarding the way the server schemas are composed.

In order to really take advantage of GraphQL's benefits, investing in a client-side technology is recommended. Client-side query merging and batching, query caching, and subscriptions are some assets that can be expected in an Apollo integration.

Configuring Apollo

Apollo's configuration demands the initialization of two different instances, *ApolloClient* and *ApolloProvider*.

ApolloClient represents the core API in a client implementation. Using this instance it is possible to configure client side caching, specific query behaviors and the the network interface. Server's endpoint and communication headers must be defined by this instance. *ApolloProvider* is responsible for attaching the *ApolloClient* to the React Native's components [42].

Chapter 4

App Development

React Native and GraphQL development practices were extracted by introspecting the processes of creation of applications with a supporting API. Most key features implemented during the development process were previously outlined so that an analysis on said technologies could be carried upon. Regarding the client side of the project, the main goal for the application was for it to work for both Android and iOS, sharing a high percentage of code. Bearing that in mind, there was a concern on selecting libraries and creating components compatible with different platforms. To prove how natively fluid the produced applications felt, aspects like the screen navigation and the way animations were handled were taken into consideration by pursuing and evaluating different solutions.

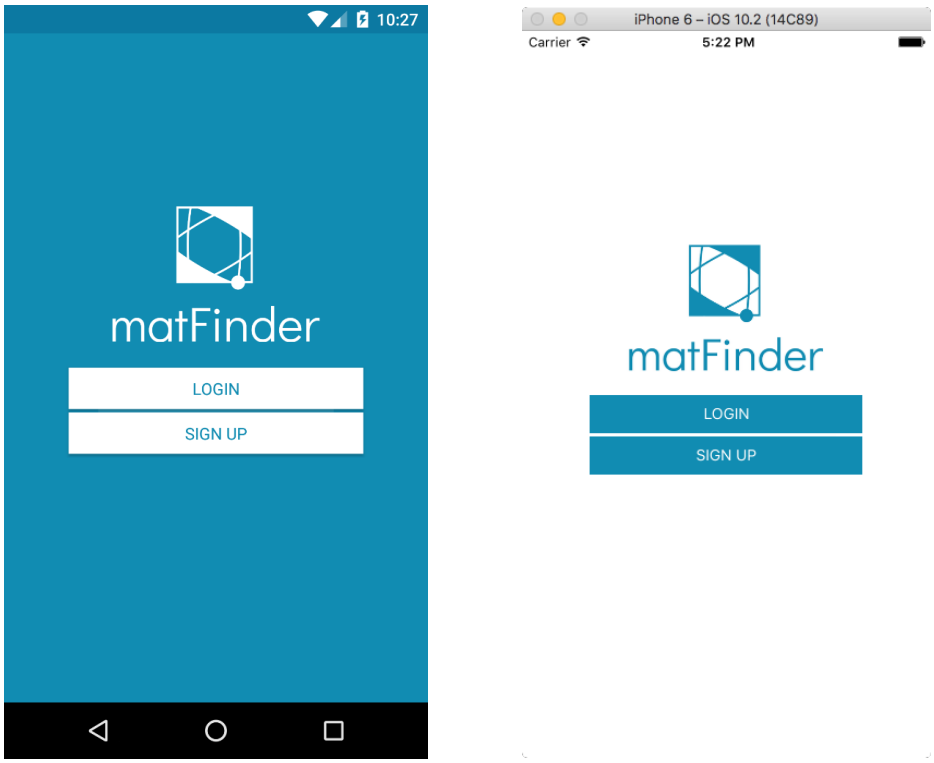
On a more technical aspect there was an evaluation of the app's performance regarding its responsiveness and the way data was fetched. The interface was designed with the intent of following most UI design principles and guidelines suggested by each platform.

GraphQL's excels in systems where a single network request can selectively load different kinds of resources at once according to a client's request. There was an attempt to explore most of the features that define these environments along with solutions to typical business-oriented operations such as authentication.

4.1 Material Finder

The main goal of the project was to extract the most information and insight from React Native development of a cross-platform application supported by a GraphQL web server.

Having that as a concern the developed application was never intended to represent a product ready for production but rather to represent how well certain key functionalities of those technologies perform when building data-driven applications. The application developed to fulfill that purpose serves as a search engine that indexes and compares properties of different materials.



(a) Android (b) iOS

Figure 4.1: Initial screen of the app on both platforms.

4.1.1 Problem

Both in academic and business environments, the task of searching/comparing different physical and chemical properties of materials consumes a great deal of time. Not only due to the huge diversity of materials but also because of the ongoing appearance of new ones. Sometimes, not having easy access to information about a property of an already studied material could turn into a painful task, forcing researchers to scroll through articles and scientific papers. Thus, although there have already been some projects in the past that attempt to deal with this problem, none have done it in a clear, uniform and mobile-friendly way.

4.1.2 Implementation

The main feature of the app is the search engine. Upon authentication, the user is able to search for materials. The results appear as a list on the main page of the application. When the user clicks on an entry a page with detailed information about the material shows up. At any time the user is able to go back to the search page or bring up the *compare* menu (which allows side-by-side comparison of materials). The system follows a client-server model where the clients are represented by each single application running on a device. This chapter describes how the development process unrolled throughout the project, ultimately contributing to the extractions of conclusions by analyzing different development practices.

4.1.2.1 Client

The developed application runs natively in both Android and iOS systems. The interface was designed taking into account the UI design guidelines of each system. React Native offers a lot of flexibility in terms of project organization. Because there isn't a *consensus* (at least for now) for the best way of organizing a React Native project, developers are given the liberty to configure their project accordingly to their needs.

Organizing the code

The first step towards the development process consisted in delineating the way the application's source code would be organized. After being familiar with the technology it was possible to determine what would bring more value to this research. There was an attempt to benefit from maximizing code reuse, expecting that it would justify having cross-platform applications. Even if certain UI components had to be displayed differently across platforms, the logic behind them should be shared.

Typically, in front-end development, lots of lines of code tend to be dedicated to configuration, from preset colors to styles, from queries to image paths. These pieces of code can be found multiple times throughout a single application and ideally should be centralized so that they're easy to modify. Configuration specifics were kept a part from the code itself.

- **android/ ios/**
  These two folders contain all the native code from each corresponding system. They encapsulate the typical files that are generated when initializing a project with Android Studio or Xcode. This way, it is possible to open the projects on the corresponding IDEs.

- **index.android.js index.ios.js**
  These two files are the entry point for each corresponding app. Because the code in this project is meant to be shared, both files point to a single file (*client.js*) and where the applications are registered (via React-Native's AppRegistry).

- **node_modules/ package.json**
  All project's external dependencies are stored inside the *node_modules* folder. The file *package.json* exists to keep track of the list of dependencies and their corresponding version. These files are automatically generated and of standard use in node environments.

- **src/components**
  The *src/components* folder is where the code that generates the views resides. Each file in this folder represents a view in terms of how it should behave and display. Some components are stateless and therefore less complex. Others might have have embedded GraphQL

queries which are responsible for handling how fetched data will display.

- **src/assets**
  All the configuration files for styles, queries, images and other settings belong in the *src/assets* folder. This folder also holds some importance regarding the cross-platform styling. Some styles are applied accordingly to the device that is running the app.

- **client.js**
  This file acts as an intermediary entry point for the applications since the code is meant to be shared across platforms. It acts mostly as a set up file as it is responsible for declaring the screens that will be shown throughout the apps' navigation. The configuration responsible for communicating with the server also lies here.

Structuring/designing the interface

The process of structuring the interface anticipated the development stage. This was accomplished by establishing a schematic of screens that would comply with the app's proposed main features. After defining the screens, they were split into components and the navigation between them was delineated.

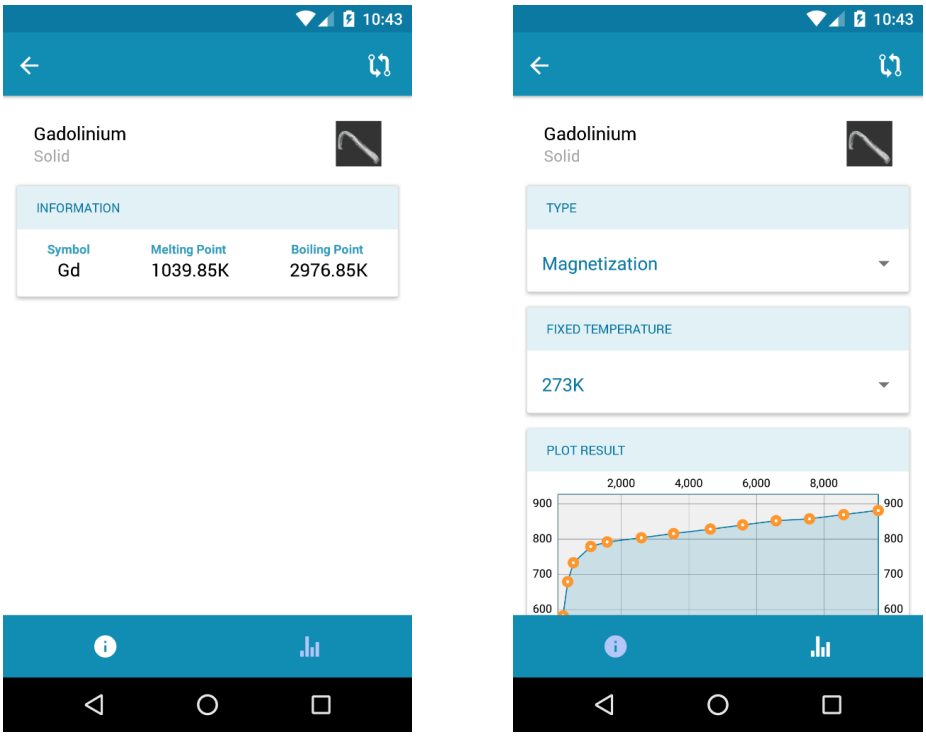
Table 4.1: Hierarchical list of components

Auth	index	
	Login	
	Signup	

Main	index	
	MaterialCompare	index
		CompareEntry
		ComparePage
		ComparePlot
		CompareResults
	MaterialPage	index
		MaterialInfo
		MaterialPlot
	MaterialSearch	index
		MaterialEntry

Navigation	*DrawerMenu*
	ActionSheet

Those screens were organized into 3 different kinds of components. Some were related to user authentication (performing login and sign up operations), others were related to navigation



(a) Info tab view.

(b) Property plot tab view.

Figure 4.2: Screen showing a page of a single material and its properties.

(like drawer menus and action sheets) and finally a group was reserved for components that would encapsulate the views for the app's main features (see table 4.1).

One of the main goals of this project was to produce natively looking applications sharing the same code. That being so, the apps were designed accordingly to each platform's user interface design guidelines. Not only visual aspects (like the color scheme or the elements' styling) were taken into account. Aspects like the way navigation menus should behave in different systems were also considered. Although it might not seem challenging, this process contributed with valuable insight for cross-platform development.

Developing the application

After having the project structured in a conceptually correct way, the development process started. The realization that working simultaneously on both iOS and Android applications wasn't efficient came on an early stage. There would always be something that needed a tweak and testing both platforms whenever something changed revealed to be time consuming. Taking that into account, and after having a couple of screens working for both platforms, the development started to prioritize Android systems. Because most components were meant to be shared, the iOS version was only engaged on the later stages of development. The work later performed on iOS was related to changes on specific UI design guidelines and related to the implementation of a native iOS library as a component.

This was the stage that allowed the extraction of most of the patterns and results. Different approaches to solve equal problems were encouraged throughout the process, specially if something felt poor on performance or *not native enough*.

4.1.2.2 Server

An HTTP server capable of interpreting GraphQL queries was developed. It was intended for it to act as a unified layer for accessing and manipulating data. The work in the server revolved around figuring out the best approaches to establish client communication through GraphQL querying. Attempts were made in order to infer good practices about authentication procedures on such environments as well as other key features related to data management. Data was stored on a MongoDB database which was populated with information from periodic table's element material types. This information was gathered using WolframAlpha's open API [43].

Choosing a GraphQL client

There are already established solutions (such as Redux [44]) that allow for GraphQL implementations. Nonetheless, most previous solutions weren't designed having in mind GraphQL's capabilities. Although it wouldn't impact this project's success, implementing a GraphQL specific client felt necessary to better understand and take advantage of those benefits. To this date, Apollo and Relay [45] both represent GraphQL clients that implement query and data fetching while optimizing requests to GraphQL servers. Despite not being identical, Apollo and Relay share most of their features. After searching and pondering about the different clients, Apollo was chosen to integrate the project. The most impactful aspect that led to this decision was the fact that Apollo offers a wider compatibility with other frameworks, compared to Relay which only offers compatibility with React and React Native environments. Since there were no other differences revealing obvious superiority of one client over, another the most flexible and generic implementation was opted by. Furthermore, Apollo's documentation is better maintained and organized which makes it ideal for someone who is starting to explore these technologies.

Developing the API

The server-side of this project is responsible for distributing an API that supports client applications. It was built using Node.js as runtime environment solely to benefit from the fact that the client side is also in JavaScript, hence bolstering code shareability. The first implementation on the server was the integration with the chosen GraphQL client, Apollo. GraphQL server implementations require a schema defining object types and their relationships.

```
type User {
  firstName: String
  lastName: String
  email: String
}

type Material {
  name: String
  phase: String
  boiling: Float
  melting: Float
  density: Float
  plot: [Plot]
  (...)
}
```

```
type Plot {
  property: String
  data: [PlotParams]
}

type PlotParams {
  t: Float
  h: Float
  y: Float
}

type Session {
  token: String
  message: String
}
```

Listing 4.1: Object types and fields.

The types were defined having in mind which and how data should be presented on the client side. For instance, a type Material was created. Each entry of this object has several fixed fields that represent unchangeable information about the material in question like its name or its boiling point. The plot visualization of each material's property is also linked as a field. Because each plot has an undefined number of parameters, there's a field in Plot consisting of an array of points destined to be rendered in a chart. In order to perform authentication a type User was created containing fields related to their personal information such as a name or an email address. Because a user's password is never intended to be retrieved through querying, this field does not belong on the type User.

To manage authentication and to control API's access a type Session was created. This type retrieves a status message alongside a token which is generated whenever a user logs into the application. This token is used by the client and sent through the headers with each request. If the token is valid the server answers the clients' request. The implementation of the described type system can be consulted on listing 4.1.

```
type Query {
  user(email: String!): User
  material(name: String!): Material
  search(term: String!, phase:String!): [Material]
  plot(material: String!, property:String): [Plot]
}

type Mutation {
  login(email: String!, password: String!): Session
```

```
10      signup(firstName: String!, lastName: String!,
11          email: String!, password: String!): Session
12    }
```

Listing 4.2: Queries and mutations.

GraphQL requires routes for queries and mutations to be defined in the schema as types. Thus, it is possible to query a user's information by providing an email; retrieve a material according to its name; retrieve an array of materials in which the names contain a term provided by a search input and so on. Only mutations related to authentication were created, one for logging in and another for signing up. Both require different parameters and both retrieve an object of type Session. The queries and mutations were declared as seen on listing 4.2.

The validation of the mutations and the logic behind the querying is done by writing resolvers. In GraphQL, each field of a query can have a resolve function telling the server how data should be returned. Having the *plot* query (from listing 4.2) as an example, the response differs depending on the *property* parameter passed. Because resolvers can return promises, as one starts to have more and different resolvers the system can become unsustainable. To avoid having to deal with difficult scenarios, an abstraction responsible for fetching data from the database was created. These functions are kept in a separate file called *connectors* and are used by the resolvers whenever data needs to be retrieved.

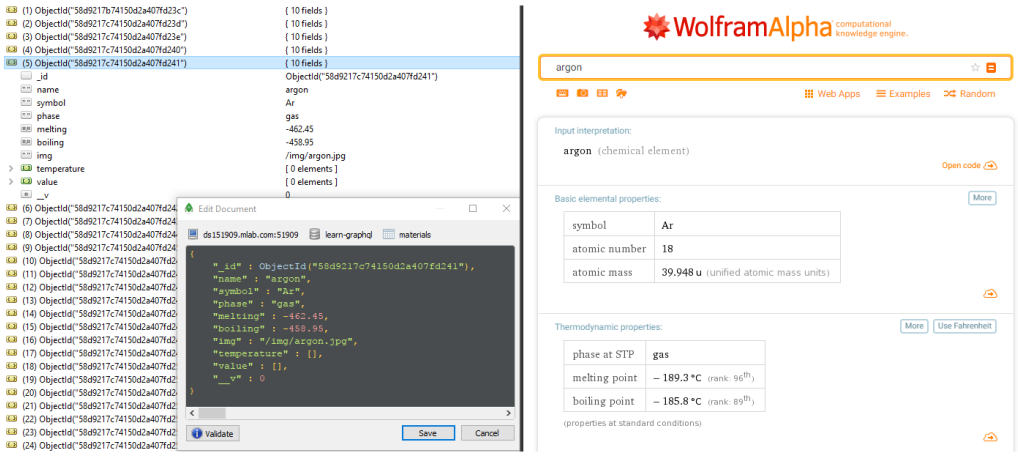


Figure 4.3: MongoDB database (*left*) and WolframAlpha's Material API (*right*).

A NoSQL database using *MongoDB* [46] was implemented in order to benefit from JSON-like documents with schemas to store information written in JavaScript (using *mongoose* [47] for object modeling). This database was initially populated through a script, also created in JavaScript, that went trough all the elements of the periodic table, queried respective information using WolframAlpha's Materials API [43] and finally inserted the parsed data into the database (see figure 4.3).

Chapter 5

Practices Analysis

During the development stage there was a continuous effort to explore new methodologies and solutions to encountered problems. This process revealed to be extremely enriching because it not only contributed to having clearer understanding about the technologies stack but also allowed the extraction of patterns, practices and some guidelines. The full-stack nature of the developed project permitted covering various and distinct aspects, from performing the initial setup to making performance adjustments.

5.1 Setup and bundling

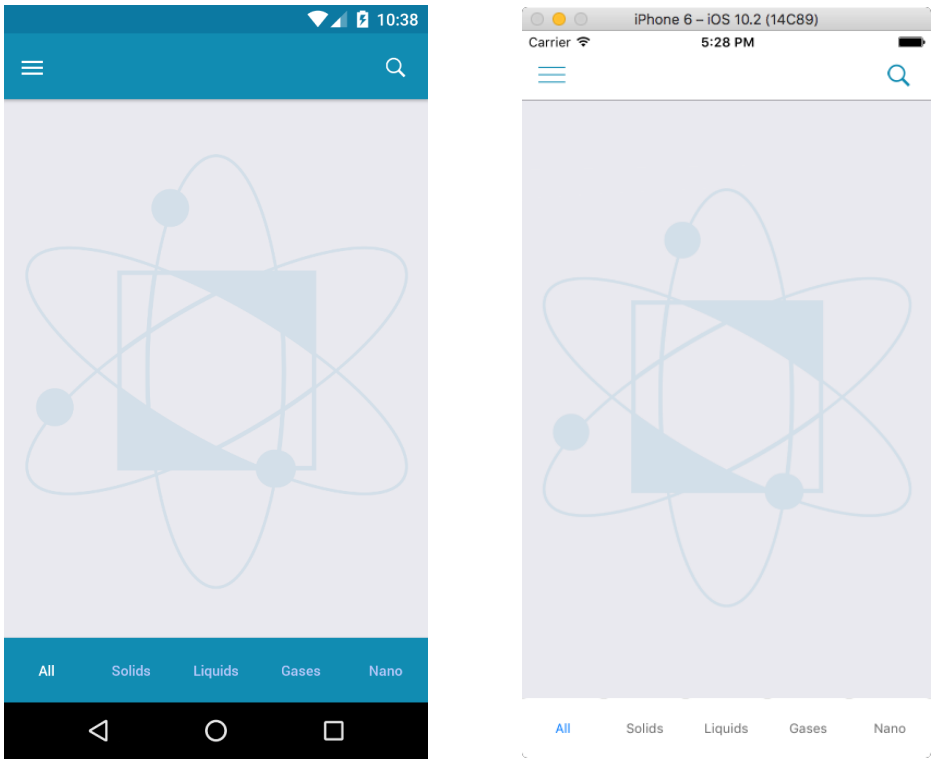
When using React in a web project developers will often have to choose a bundler and decide which bundle modules are needed. Contrarily, React Native's framework handles bundling inherently, allowing the developer to quickly set up a project without having to worry, for example, about setting up a JavaScript transpiler to code accordingly to ES6 or ES7 standards.

Depending on the development operating system and the targeted platforms, React Native requires different dependencies to be installed. There's a *getting started* guide on the framework's main page. For instance, on Windows only Android development is possible while on macOS it is possible to target both platforms.

Besides the different dependencies of each operating system it's also mandatory having Xcode and the Android SDK installed along with the respective emulators.

After everything is set up, it is possible to use npm [31] to globally install React Native's command-line client. That client will then allow creating and running new RN projects in a fast and simple way.

A crucial factor for the success of the project was having a carefully maintained Git [48] repository. To minimize compatibility problems between platforms it is important that platform specific code is kept apart from generic code. Additionally, the repositories should ignore changes on binary files which result from building/compiling the applications.



(a) Android (b) iOS

Figure 5.1: Initial screen of the app on both platforms for authenticated users.

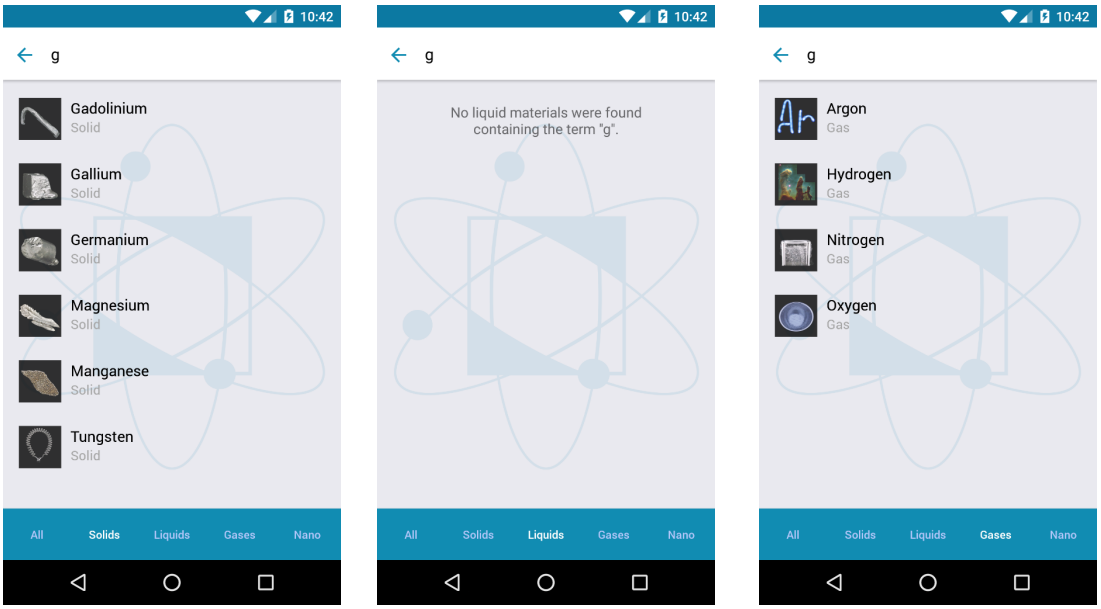
5.2 Important components

As it was briefly stated on section 4.1.2.1 the components were organized in 3 main groups. The first group of components contains views related to user authentication. There's also a group of components which contain views related to navigation such as drawer menus (Android) and action sheets (iOS). The rest of the components belong to the application's main views. In this section, relevant implementation details for each component are examined.

5.2.1 MaterialSearch

MaterialSearch is a component that implements the main feature of the application. This component has a navigation bar at the top with two buttons. The leftmost opens a menu dialog and the button on the right opens a text input form, which the user will use to perform material searches. When performing a new search the component will detect that a user stopped typing and only then will query the server passing the search term as a parameter.

As seen on figure 5.2 it is possible to arrange the materials (search results) according to their physical phase at room temperature. This can be achieved by clicking on the tabs located at the bottom of the screen. Whenever the user changes tabs or edits the search terms a change in the state of the component will trigger automatically, refetching new data.



(a) Solids (b) Liquids (c) Gases

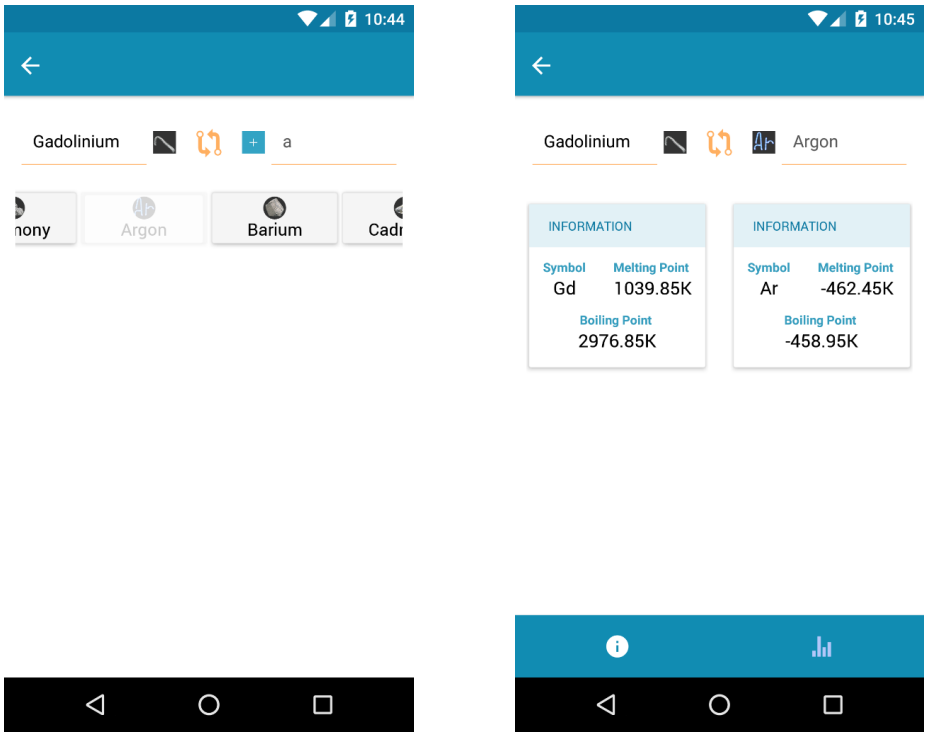
Figure 5.2: Switching between tabs while searching for materials.

Initially, the results were rendered using React Native's ListView [49], a core component designed for displaying vertically scrolling lists. Each row of the list represents a component (called *MaterialEntry*) associated to a material. Even though the results were being correctly rendered the performance was lacking, specially when the number of materials surpassed the number of visible entries at a single time. Ultimately, this was fixed by changing the ListView component to a FlatList [50], another core component for rendering lists that was recently released with RN v0.43. The process of improving this component's performance is better described in section 5.7.

5.2.2 MaterialCompare

The application contemplates a feature that compares physical properties between different materials (see figure 5.3). There are two ways of accessing this functionality, from the main screen via the menu or by hitting the top right corner button of a material's page (see figure 4.2). When accessing through a material page, the user will find a text input where he will select the other material for comparison.

This functionality was thought out and implemented using GraphQL fragment queries thereby combining two UI components requirements into one initial data fetch (see listing 5.1) that returns a single response (see listing 5.2). This represents a situation where existing components are reused. The same components that render a single material page are used on the comparison screen but horizontally split into two views.



(a) Selecting materials to compare.

(b) Comparing two different materials

Figure 5.3: MaterialCompare component.

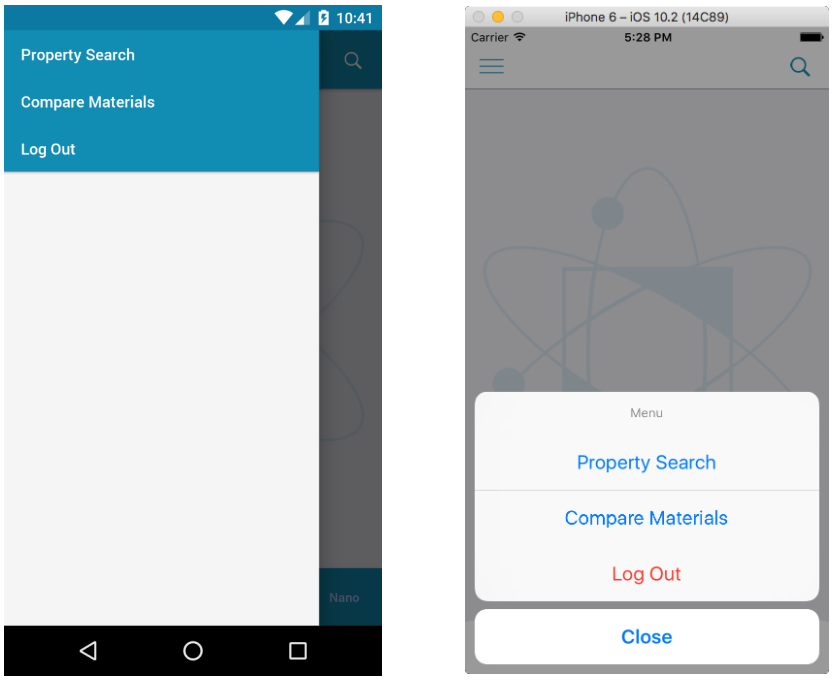
```
query {
  firstMaterial:
      material(name:"gadolinium"){
        ...info
      }
  secondMaterial:
      material(name:"argon"){
        ...info
      }
}

fragment info on Material {
  name
  img
  symbol
  boiling
  melting
}
```

Listing 5.1: Fragment query.

```
{
  "data": {
    "firstMaterial": [{
      "name": "gadolinium",
      "img": "/img/gadolinium.jpg",
      "symbol": "Gd",
      "boiling": 2976.85,
      "melting": 1039.85
    }],
    "secondMaterial": [{
      "name": "argon",
      "img": "/img/argon.jpg",
      "symbol": "Ar",
      "boiling": -458.95,
      "melting": -462.45
    }]
  }
}
```

Listing 5.2: Fragment response.



(a) Android (b) iOS

Figure 5.4: Initial screen of the app on both platforms.

5.2.3 Menus

The developed application has few screens to inter-navigate so the user should be able to use the app without accessing the menu. Regardless, most applications need to implement some sort of menu display and, for that reason, the same was done in this project. However, there are many ways of dealing with menu dialogs and each system uses different user experience guidelines.

On Android it is suggested that menus are handled using drawer navigation [51] while iOS's guidelines are action sheets [52] advocates. One approach to implement the menu dialog could be chosen and implemented in both platforms but the end result wouldn't be as natively fluid as it should. To better explore this dilemma both approaches were implemented (see figure 5.4). This way, user experience is improved at the cost of having less code shared between platforms.

5.3 Navigation

React Native's documentation provides developers with several components to handle navigation. When this project's development stage started, RN was on version 0.41 [53] and its documentation suggested using a component called Navigator which provided a JavaScript implementation of a navigation stack compatible with iOS and Android. Due to being a constantly growing relatively recent technology, RN often changes its documentation and implements new components. During the development stage the Navigator component got deprecated and, for that reason, a major

change on the code base was required. Since version 0.44 [54], RN recommends using a library called React-Navigation [55].

This event turned out to be enriching for the project because it mindfully gave insight about the amount of work needed whenever something gets deprecated and needs to be re-implemented.

Refactoring, introducing a new navigation pattern and implementing a new navigation state management are tasks that a developer should not be forced to do in later stages of development. Fortunately, this solution appears to be a final and official solution.

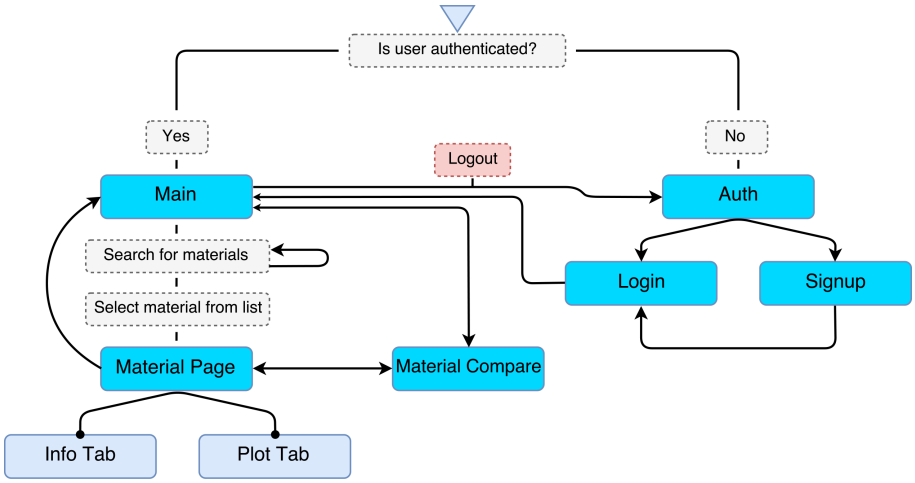


Figure 5.5: App navigation flow diagram.

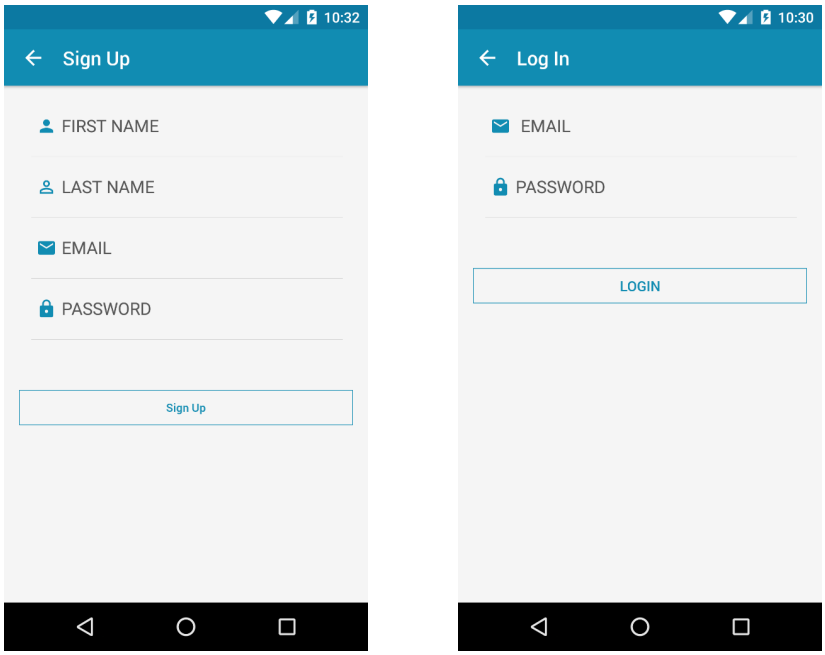
The different screens and components for navigation are defined in the entry file that is used by both platforms. Information can be transfered between screens by defining each screen's *navigationOptions*. For instance, each material page screen is a component that should receive information about the material that will be displayed.

Most of this refactoring process was achieved with minor changes. The most challenging aspect of this process was reintegrating the Apollo GraphQL client. Although it was not immediate, the solution to this problem was achieved by creating an auxiliary function that would wrap the GraphQL client provider around the screen components.

5.4 Authentication

Authentication is handled on the client side in two different views. These views can only be accessed (from the main screen shown on figure 4.1) if the user is not already authenticated in the application. If that is the case two buttons are shown; one of them will bring up a new screen with a login form (for already registered users) and the other will show a sign up screen.

Both components are exported with an embedded mutation. Whenever the input fields are validated the submit button becomes enabled. Once clicked, the client will contact the GraphQL server by passing the value of the input fields alongside the mutation. The server will then validate the request and respond accordingly. The response will vary depending on the parameters passed.



(a) Sign up (b) Login

Figure 5.6: Authentication screens for Android.

A token will only be sent if the email and password match the data from the database. Otherwise an error message will be sent (*"User not registered/Wrong password"*). If a token is retrieved the client will store it locally using RN's AsyncStorage [56].

The Apollo client network interface has a middleware feature that allows adding authorization headers to HTTP requests. So, using that middleware, it is possible to pass the token through the headers on following requests ensuring the user's authenticity.

```
storeToken = async (token) => {
  try {
    await AsyncStorage.setItem('@MatFinder:token', token);
  } catch (error) {
    console.error(error)
  }
}

handleLogIn = async () => {
  const { email, password } = this.state
  const { navigate } = this.props.navigation
  this.props.mutate({ variables: { email, password } })
    .then(({ data }) => {
      /* if(SUCCESS) { */
        this.storeToken(data.login.token)
        let user = decoder(data.login.token).data
        navigate('Main', { user })
      /* } else if(ERROR)  */
```

```
19            this.setState({
20              error: true,
21              message: data.login.message
22            })
23          })
24    }
```

Listing 5.3: Code excerpt from Login component class to handle authentication and token storage.

GraphQL specification does not declare how to handle authentication so developers are given the liberty to implement it according to their specific needs. In this project a new GraphQL type was created for session handling. This type called Session (see listing 4.1) contains two fields, one to hold the authorization token and another to send a status message, and is sent whenever a user performs a login or sign up operation.

5.5 Animations

The animations throughout the application are handled using React Native's Animated library [57]. This library was designed to perform fluid animations while having declarative relationships between inputs and outputs, with methods to control time-based execution. Most implemented animations were basic fade-in, fade-out transitions and existed to please and enhance user experience.

In order to explore React Native's Animated library potential, an attempt to replicate Gmail's app [58] search bar composed animation was carried out. Because this library makes use of native drivers to perform animations it should be possible to achieve the same feeling that a native app would. The first step to replicate this consisted in observing the way the screen behaved to touch. Once the magnifying glass icon was clicked the main navigation bar would fade out and instants later the search form would show up. Taking those observations into account it seemed obvious that at least two groups of elements had to be synced. This synchronization was achieved by setting a boolean variable to the main component state that would indicate whether the menu search bar was active or not. Having that set it was only required to configure the animations. At first, a simple approach was attempted; whenever the search was meant to show up, the navigation bar opacity would be incrementally lowered until it was hidden while the search form would do the opposite until it appeared on screen. Unfortunately, the end result didn't feel natural because when both elements were at half opacity they would overlap each other (as they were occupying the same spot on the screen). So, in order to fix this, parameters related to the elements' position also needed to be changed.

Ultimately, whenever the search button was pressed the navigation bar would gradually fade away while also going off screen (by decrementing its height). Concurrently, the search form would do the opposite and show up on screen. The increment and decrement timers were adjusted so that they'd match Gmail's app behavior.

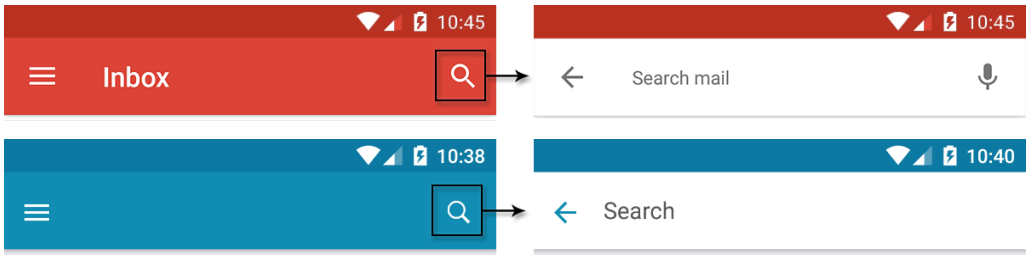


Figure 5.7: Gmail's search bar animation (*top*) and its replication on the developed app (*bottom*).

```
1   Animated.parallel([
2     Animated.timing(
3       this.state.searchOpacity, {
4         toValue: 1,
5         duration: 1
6     }).start(),
7
8     Animated.timing(
9       this.state.searchHeight, {
10        toValue: 60,
11        duration: 1
12    }).start(),
13
14    Animated.timing(
15      this.state.navHeight, {
16        toValue: -60,
17        duration: 1
18    }).start()
19  ])
```

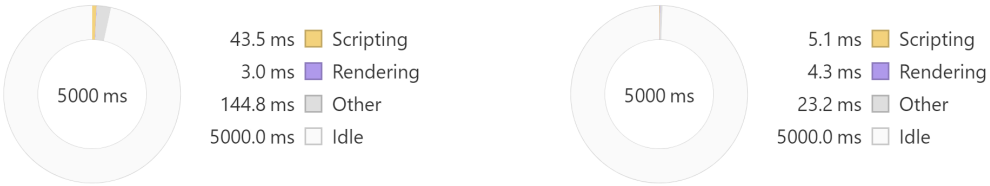
Listing 5.4: Composed animation triggered when search bar is displayed.

```
1   Animated.parallel([
2     Animated.timing(
3       this.state.searchOpacity, {
4         toValue: 0,
5         duration: 1
6     }).start(),
7
8     Animated.timing(
9       this.state.searchHeight, {
10        toValue: 0,
11        duration: 1
12    }).start(),
13
14    Animated.timing(
15      this.state.navHeight, {
16        toValue: 0,
17        duration: 1
18    }).start()
19  ])
```

Listing 5.5: Composed animation triggered when search bar closes.

5.6 Data fetching

The applications were developed so that each component could be responsible for the data needed to be fetched. Using Apollo it is possible to attach GraphQL query results to React's UI. This is achieved by wrapping components with queries, thereby declaratively specifying the structure of the view. Rather than fetching data from *componentDidMount()* (as React Native's documentation suggests) Apollo fetches the data when the component is at play. When a component is decorated with a query, Apollo will pass certain properties to the components that help handling data fetching states such as information about whether an error occurred or if the data is still loading. The application uses those properties to handle errors while displaying convenient messages. While



(a) ListView (b) FlatList

Figure 5.8: Measuring elapsed time when performing the same operation in different components.

the data is being fetched the app displays a loading spinner. When done, the loading state will change and the spinner will be replaced with data filled views.

5.7 Improving performance

While the development stage took place there was always an extensive concern on having a clean and fast interaction within all sections of the application. There were three non-related issues that needed improvement and were successfully corrected. Gladly, React Native is aware that there are still performance issues and provides a page with some tips on how to fix them.

As briefly stated on section 5.2.1, at first the search results where being rendered in a ListView component. However, this component was performing poorly when it had to render large lists. The initial render time was improved by setting values for properties like *initialListSize* and *scrollRenderAheadDistance*. Although the initial rendering was fluid, there was still a performance issue with the screen transitions. There was still a noticeable and irregular delay when an entry of the list was clicked. After doing some research it was concluded that the ListView was causing an excessive memory consumption. The issue was acknowledged by Facebook [59] which then released an alternative component, called FlatList, to replace the soon to be deprecated ListView. Having implemented the new FlatList component on the app, the abnormal transition delays were no longer existent. The problem was fixed because, with FlatList, items outside of the render window are unmounted and their memory is reclaimed avoiding the memory usage fluctuation that was causing inconsistency.

The app's performance was monitored to better evaluate the improvement from changing components. In order to evaluate which component performed better, Chrome's profiler [60] was used to monitor activity while debugging. Although the results are not accurate they are sufficient to help decide which approach performs better. The results that proved that FlatList had better performance over the ListView component can be consulted on figure 5.8.

Both implementations of the components where given a timespan of five seconds where they queried the server, received the results and displayed them on the screen. The operation was performed almost six times faster using the FlatList component in comparison to the ListView component. It is important to notice that these results were extracted from the app while running on a virtual emulator. If they were ran on a mobile device, because the computational power tends to be lower, the performance times would be aggravated.

5.8 Styling

Currently there are three main UI libraries for React Native development, *Shoutem UI* [61], *React Native Elements* [62] and *Native Base* [63]. These libraries help streamlining the design of applications by providing beautifully styled components ready to be integrated. Native Base builds their components to match the appearance of native components while following the UI design guidelines of each platform. Contrarily, *Shoutem* and *React Native Elements*, implement their own uniquely styled components. Because *Native Base* was designed over platform recommendations, this was the chosen library that helped expedite the design of a fully cross-platform app. Using *Native Base*, developers are able to pass different style properties to the components as well as callback events.

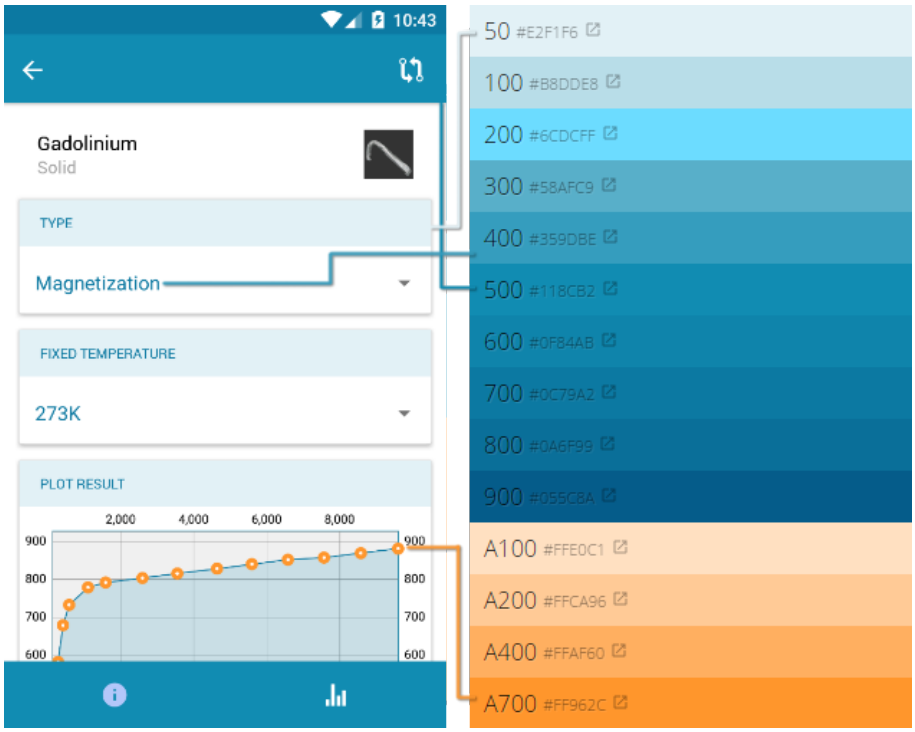


Figure 5.9: Color palette defined for Android platforms.

React Native presents an abstraction similar to CSS StyleSheets [64] to help the customization of the style properties of components. Using this abstraction the styles can be defined outside the render function, tidying up the code hence improving code quality.

Native Base library contributed to code shareability as its components respect certain platform rules like enforcing that each platform uses standard typography and icons. Nonetheless, there were other aspects that needed configuration. An example was the definition of the color schemes which differentiate a lot between platforms. While iOS follows a minimalist approach which suggests using color judiciously for communication [65], Android suggests defining a strict color palette filled with primary and accent colors [66] (which was implemented as seen on figure 5.9).

Both applications share the same primary color which is applied differently depending on the running platform as can be noticed on figure 5.1.

5.9 Using native modules

Having a vector icon library was seen as important in order to make the application more aesthetically pleasing. Because icons follow standards according to the running platform, a search for a native module that would provide a library with cross-platform SVG icons was carried out.

Another search was carried in order to find a native module that would perform plotting and render charts. It was intended for the charts to be dynamic and to perform as fast as possible so, because the native thread has priority over the JavaScript thread, libraries running native modules were a priority.

There were many possibilities but, in the end, the chosen modules were *React Native Vector Icons* [67] (the highest rated library for cross-platform icon rendering) and *React Native Charts Wrapper* [68] (a module that natively renders charts using cross-platform ready components, supporting both Android and iOS).

To start using these modules they had to be firstly installed as *Node Package Manager* dependencies. For instance, to install *React Native Vector Icons* dependencies, the command **npm install react-native-vector-icons** was issued using the terminal on the project's directory. After the installation was complete it was possible to link the native files from the icon library to both Android and iOS projects. This linking process can be done manually or automatically using React Native's command line and issuing the command **react-native link react-native-vector-icons**. The second approach was pursued to avoid messing with native code.

Having downloaded the dependencies and linked the native code files it was possible to start using the libraries by importing them on the components files.

5.10 Platform specific code

The produced code is shared between platforms whenever possible. The only scenarios where the code differentiates occur to ensure that both interfaces follow each platform's design guidelines.

React Native provides a module that helps managing platform-oriented code. This module can detect the platform in which the application is running. Table 5.1 shows the amount of generic lines of code versus platform specific lines of code for each component. Upon consultation, it's possible to say that few of the developed components required having platform-oriented code. The ones that did, only implement platform detection logic to fit styling specification needs or to integrate platform specific components (namely the *Navigation* components).

```
render() {
  let user = this.props.navigationOptions.user

  return (
    (Platform.OS === 'android')
    ? ( <DrawerLayoutAndroid /*(...)*/ >
        { this.renderContent() }
      </DrawerLayoutAndroid> )
    : this.renderContent()
  )
}
```

Listing 5.6: Code excerpt from Main component class to determine which navigation component should be rendered according to OS.

The React component that wraps the platform DrawerLayout [69] from Android used for displaying the drawer menu demands the main view components to be its direct children. Because this component is only meant to be run in Android devices, only a small section of the Main component needs to be platform specific (see listing 5.6).

Table 5.1: Relation of platform specific lines of code between components.

Component		**Generic LoC**	**Platform Specific LoC**
Auth	*index*	138	7
	Login	180	0
	Signup	191	0
Main	index	217	27
MaterialCompare	index	146	0
	CompareEntry	57	0
	ComparePage	137	0
	ComparePlot	394	21
	CompareResults	135	0
MaterialPage	*index*	124	0
	MaterialInfo	68	0
	MaterialPlot	318	25
MaterialSearch	*index*	84	0
	MaterialEntry	54	0
Navigation	*DrawerMenu*	35	35
	ActionSheet	58	58

Total:	2336
Platform Specific:	173
Shared Code:	*~99.93%*

5.11 Developer tools

5.11.1 Visual Studio Code

Visual Studio Code (also known as VSCode) [70] is a source code editor developed by Microsoft. It is supported on Windows, Linux and macOS systems. This editor was designed without having a target platform in mind and because of that it is recommended for cross-platform development. It is highly customizable and brings inherent features like debugging, Git control, syntax highlighting, code completion and refactoring.

VSCode is free, open-source and can be extended via plug-ins. An extension called React Native Tools [71] supported the development stage of this project by providing code-hinting, debugging and integrated commands for RN.

5.11.2 Ngrok

Ngrok [72] is a tool that allows creation of secure introspectable tunnels to localhost. During the setup stage of the project it was noticed that the device simulators were not being able to connect to the web server that was running locally (React Native kept throwing the error : "*Could not connect to development server*"). By letting Ngrok listen to the port where the web server was running, a URL was provided that could be accessed by any simulator with an internet connection.

5.11.3 GitHub

GitHub [73] is a web-based Git repository hosting service. With more than 26 million users, GitHub represents the largest host of source code in the world.

Git consists in a command-line tool used for version control which was developed by Linus Torvalds, creator of Linux OS, in 2005. Git allows developers to periodically commit their code granting them the ability to track changes. After a commit, Git takes a snapshot of the code and stores the differences from previous commits. Developers can keep track of the changes in any set of files and even roll back changes to any stored versions.

Integrating version control on this project was important as it encouraged experimentation and exploration of different development methods without having the risk of compromising the code (because changes could always be reverted with little effort).

Chapter 6

Validation

This chapter sums up and discusses the extracted development practices and patterns presented earlier in chapter 5. It explains and reviews the development concerns that were taken into account to meet the proposed goals.

6.1 Setup and bundling

Over nine thousand developers took part on a annual survey, regarding the state of JavaScript, that covered topics regarding application development [74]. Last year's edition concluded that native applications are still the most common solution when it comes to mobile development. According to a sample of 9307 developers, React Native had a 92% satisfaction rating and a 78% interest-to-learn rating. The last result shows that there are a lot of newcomers willing to embrace this technology. For this reason, the insight given on the initial project setup and bundling (see section 5.1) is considered valuable information.

6.2 Implementing components

Every project has different requirements; therefore, in React Native, the integrating components of different projects will also have differences. To provide a broader range of principles regarding component development, three of the implemented components were analyzed. Each of them tries to demonstrate a principle related to the data-driven aspect of the developed application.

The main component of the application is described on section 5.2.1; due to a visible lack of performance, a thorough examination of its children components was conducted and portraited on section 5.7. This examination process was a success as the problem was rectified. Situations like this can often occur; it is important to know how to figure out the root of the problem and how to fix it.

The implementation of the *MaterialCompare* component (see section 5.2.2) is referenced to emphasize and inform about the advantages of GraphQL in data-driven applications. This component embeds on its view a GraphQL fragment query that is able to perform multiple interrogations with a single request.

The last mentioned component implementation regards the implementation of platform-specific components affirming the advantages of doing so as a way to achieve better UIX. Additionally, insight on how to approach this scenarios is given on section 5.10. Code shareability is encouraged by React Native; it was proven that it is possible to achieve a high percentage of shared lines of code (more than 99.9%) even when some components implement platform specific features (Table 5.1).

6.3 Styling and animation

Another goal for this project consisted in having applications that looked and felt native. React Native's Animated library ensures that the interpolation of the animations is executed natively. To explore this feature an implementation of Gmail's app search bar animation was attempted. This process of replication is described and demonstrated in section 5.5.

As for the styling, both the Android and the iOS applications tried to follow each system's UI design guidelines. The final applications follow several design guidelines aspects such as respecting color schemes, specific layout placements and platform standard icons. Accomplishing these requirements contributed highly to help determine how much code could be shared without affecting the visual standards of the platforms.

6.4 Implementing authentication

GraphQL's specification doesn't declare how authentication should be handled. Therefore, the implementation of the authentication processes presented in this thesis is a valid contribution as guideline. Common practices of session handling in React Native are also demonstrated (see section 5.3). The whole authentication system was implemented so that only logged in users can make GraphQL queries. Users' information is safely stored in a database with the encryption of more sensitive fields. Validation codes or error messages are sent during the authentication process. The accomplished work fulfills with success most of the authentication management requirements for common applications.

6.5 Handling navigation

There are several available approaches to handle navigation in React Native applications. This project provides guidelines regarding the deprecation of existing modules and emphasizes the necessity of keeping up to date with technology changes. Navigation was handled using a methodology that was introduced by React Native's development team when the project was already in

an advanced stage. React Native provides its own guidelines for handling navigation; nonetheless, this project demonstrates a working practice that allows integration with the Apollo GraphQL client (see section 5.3).

6.6 Mobile design patterns

Common mobile design patterns were taken into account and put into practice. These patterns focus on several aspects of mobile development and can be divided into three groups: interaction patterns, presentation patterns and behavioral patterns. While some of these patterns were ensured by the technology stack, others required custom implementation.

Interaction patterns

- **Back-and-Save**
  This pattern suggests that input screen data should be saved when the user leaves the screen. The produced application interacts this way. Upon searching and clicking on a material's page the user will change screens; the user can then go back to the search screen and find the same input data and fetched results. This functionality is inherently provided by React-Native's framework which saves screen states during navigation.

- **Guess-Don't-Ask**
  This pattern suggests that users should avoid user input, specially when writing text. To minimize the effort of typing, the search functionality was developed to automatically fetch search results as the user types. This way, the users never need to type the name of a material to its full extent because it will eventually show on screen before that happens. This feature was implemented using React Native's TextInput component, that features a handler called *onChangeText*, to trigger data fetching.

- **A-la-Carte-Menu**
  This pattern suggests that users should know all the actions and options available at any time. This was taken into consideration while designing the interfaces and was successfully implemented.

- **Sink-or-Async**
  This pattern suggests that operations that last more than one second should be asynchronous. The only operation that can last longer than that is the data fetch. Apollo GraphQL client ensures that this fetch is always asynchronous. While the data is being fetched an animated ActivityIndicator [75] component is displayed to inform the user that an operation is in progress.

- **Logon-and-forget**
  This pattern suggests that user credentials should only be asked once. This was implemented

in the authentication process; if a previous token is stored locally, it will be retrieved and validated by the server; if the server confirms that the token is valid the user will not be prompted to login.

Presentation patterns

- **Do-as-Romans-Do**
  This pattern suggests using the recommended look-and-feel guidelines of each platform. This was a main concern during the whole project and was achieved on both Android and iOS platforms.

- **List-and-Scroll**
  This pattern suggests using lists with vertical scrolling for presenting results. The main component of the application shows the results of every performed search precisely in a list with vertical scrolling.

Behavioral patterns

- **Memento-Mori**
  This pattern suggests that relevant state and info should be saved whenever the application goes into background. This functionality is inherently provided by React-Native's framework.

6.7 Discussion

In order to justify the adoption of React Native and GraphQL development, when building cross-platform data-driven applications, different topics were analyzed during pattern extraction. Those topics present relevant matters to developers who are initiating or using these technologies. The validation of the extracted practices was achieved by performing an introspection focused on common application development issues.

There are numerous methodologies and practices in the mobile development environment. The ones presented and explored in this thesis have different natures. For instance, some of the practices are solely related to the integration and interaction between the two main technologies. Others focus on the visual side of applications emphasizing aspects like animations or styling. Also, and because these technologies still don't have standards for every development matter, practices regarding the implementation of common business logic procedures such as authentication were taken into account.

It is important to notice that most of the extracted practices favor aspects like code shareability and having native looking UIs. If a developer wanted to disregard those constraints some practices could become void. Nonetheless, the end result was compliant with the initial set of ideals.

To sum up, it is believed that the development phase succeeded in terms of allowing the exploration of a broad range of concerns. It was evidenced that many common mobile design patterns are inherently taken care of by React Native and GraphQL but some still require custom implementation by the developers.

Chapter 7

Conclusions

Cross-platform solutions introduce a new point of view regarding mobile development. Developers have to balance several aspects if they plan to maximize their apps' reach. Although native development often outputs better and more fluid applications, there are other kinds of constraints that deserve examination.

Native development does not follow a uniform and consistent set of rules or languages, varying from platform to platform, which forces enterprises to contract more experienced and, consequently, more expensive developers. There are few cross-platform approaches that promote code reusability and also produce natively compiled code. React Native appeared on the market promising the best of both worlds.

Typical solutions for mobile development associate applications to the functionalities of a web service. Most web services follow a REST approach which, in the current state of the art, is filled with too many formalities and specifications that impair the clearness and transparency of how systems should be developed. GraphQL promotes an easier and more efficient way to manipulate data, bringing advantages which might accelerate the development processes.

Using GraphQL and React-Native, a cross-platform data-driven application was developed. A study on its implementation was carried up, expecting the extraction of some development practices.

7.1 Accomplishments

The objectives for this dissertation are considered as achieved. The literature review contributed to learning fundamental concepts about the purposed technologies and also allowed for a deeper understanding of the current state of mobile development.

The development stage of this project resulted in a fluid application compliant with two operating systems, Android and iOS, supported by a web-server and an API thought out to enhance data-fetching. The application is capable of adapting its UI to match the defined UI guidelines of each platform and shares more than 99.9% percent of the code (between platforms).

Ultimately, this whole process allowed the gathering of a set of development practices which were presented in a clarified and detailed way.

7.2 Future Work

Both React Native and GraphQL are considered new emerging technologies in the software development scene. For this reason most problems still lack well established methodologies and patterns to use as guidelines. A continuous work to explore different approaches to find better solutions must be actively done as the software and its community grows.

Having more insight on how to write custom native modules must be a valuable asset while working with React Native. Although the topic was not explored during this thesis, as it would require focusing more on native development, it definitely deserves future examination.

The developed application was thought out to help explore the most of its supporting technologies. Notwithstanding, publishing the application on a application store could be interesting as it would require further investigation like verifying if the apps are compliant with both stores standards/requirements.

References

[1] Sarah Allen, Vidal Graupera, and Lee Lundrigan. *Pro Smartphone Cross-Platform Development: iPhone, Blackberry, Windows Mobile and Android Development and Distribution*. Apress, 2010.

[2] Jakob Iversen and Michael Eierman. *Learning Mobile App Development: A Hands-on Guide to Building Apps with iOS and Android*, chapter 16. Addison-Wesley, 2014.

[3] IDC: Smartphone OS Market Share. http://www.idc.com/promo/smartphone-market-share/os. Accessed: 27-01-2017.

[4] Jeff McWherter and Scott Gowell. *Professional Mobile Application Development*. Wrox, 2012.

[5] Rajinder Singh. An Overview of Android Operating System and Its Security Features. *Journal of Engineering Research and Applications*, 4(2):519–521, February 2014.

[6] Andrei Frumusanu. A Closer Look at Android Runtime (ART) in Android. http://www.anandtech.com/show/8231/a-closer-look-at-android-runtime-art-in-android-l/, Jul 2014. Accessed: 28-01-2017.

[7] Android Studio the Official IDE for Android. https://developer.android.com/studio/index.html. Accessed: 29-11-2016.

[8] Reto Meier. *Professional Android 4 Application Development*. Wrox, 2012.

[9] What's new in iOS. https://developer.apple.com/library/content/releasenotes/General/WhatsNewIniOS/Articles/iOS10.html. Accessed: 30-11-2016.

[10] iOS Technology Overview. https://developer.apple.com/ library/content/documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/Introduction/Introduction.html. Accessed: 30-11-2016.

[11] Start developing ios apps (swift): Jump right in. https://developer.apple.com/ library/content/referencelibrary/GettingStarted/DevelopiOS-AppsSwift/. Accessed: 03-02-2017.

[12] C.P Rahul Raj and Seshu Babu Tolety. A study on approaches to build cross-platform mobile applications and criteria to select appropriate approach. *India Conference (INDICON), 2012 Annual IEEE*, pages 625–629, December 2012.

[13] Bhushan S. Thakare, Dhanashree Shirodkar, Naghma Parween, and Shama Parween. State of Art Approaches to Build Cross Platform Mobile Applications. *International Journal of Computer Applications*, 107(20):22–23, December 2014.

[14] Sencha Touch, Cross-platform Mobile Web App Development Framework for HTML5 and JS. https://www.sencha.com/products/touch. Accessed: 29-11-2016.

[15] PhoneGap, PhoneGap Docs. http://phonegap.com/. Accessed: 29-11-2016.

[16] Xamarin, developer center. https://developer.xamarin.com. Accessed: 29-11-2016.

[17] Gustavo Hartmann, Geoff Stead, and Asi DeGani. *Cross-platform mobile development*. Tribal, 2011.

[18] Adobe AIR. http://www.adobe.com/se/products/air.html. Accessed: 30-11-2016.

[19] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed systems: Concepts and design*. Addison-Wesly, Fifth edition, 2012.

[20] Roy Thomas Fielding. *Architectural styles and the design of network based software architectures*. PhD thesis, University of California, Irvine, 2000.

[21] Leonard Richardson and Sam Ruby. *RESTful Web Services*. O'Reilly Media, Inc., First edition, 2007.

[22] Leonard Richardson. *RESTful web services*. O'Reilly, Farnham, 2007.

[23] Protocol Buffers | Google Developers. https://developers.google.com/protocol-buffers/. Accessed: 31-01-2017.

[24] Java Object Serialization. https://docs.oracle.com/javase/8/docs/technotes/guides/serialization/. Accessed: 31-01-2017.

[25] Extensible Markup Language (XML). https://www.w3.org/XML/. Accessed: 31-01-2017.

[26] Introducing JSON. http://www.json.org/. Accessed: 31-01-2017.

[27] Sebastian Bittl, Arturo A. Gonzalez, Michael Spahn, and Wolf A. Heidrich. Performance Comparison of Data Serialization Schemes for ETSI ITS Car-to-X Communication Systems. *International Journal on Advances in Telecommunications*, 8(1):48–58.

[28] Stack overflow developer survey 2017: Most popular technologies. https://insights.stackoverflow.com/survey/2017. Accessed: 19-06-2017.

[29] Stephen Cass. The 2016 top programming languages, Jul 2016.

[30] Nicholas C. Zakas. *Understanding ECMAScript 6: The Definitive Guide for JavaScript Developers*. No Starch Press, 2016.

[31] npm - building amazing things. https://www.npmjs.com/. Accessed: 19-06-2017.

[32] Bonnie Eisenman. *Learning React Native: Building Native Mobile Apps with JavaScript*. O'Reilly Media, 2015.

[33] Android studio the official ide for android. https://developer.android.com/studio/index.html. Accessed: 29-11-2016.

[34] Akshat Paul and Abhishek Nalwaya. *React Native for iOS Development*. Apress, 2015.

[35] React Native: Components and Props, howpublished= https://facebook.github.io/react/docs/components-and-props.html, journal=React Native, note = Accessed: 19-06-2017.

[36] React Native: State and Lifecycle. https://facebook.github.io/react/docs/state-and-lifecycle.html. Accessed: 19-06-2017.

[37] React Native: Component Lifecycle. https://facebook.github.io/react/docs/react-component.html. Accessed: 19-06-2017.

[38] Eric Masiello. *Mastering React Native : leverage frontend development skills to build impressive iOS and Android applications with Native React*. Packt Publishing, Birmingham, UK, 2017.

[39] Graphql. http://facebook.github.io/graphql/. Accessed: 03-02-2017.

[40] Samer Buna. *Learning GraphQL and Relay*. Packt Publishing - ebooks Account, 2016.

[41] Falco Nogatz and Dietmar Seipel. Implementing GraphQL as a Query Language for Deductive Databases in SWI–Prolog Using DCGs, Quasi Quotations, and Dicts. pages 42–56, September 2016.

[42] React Apollo | Apollo React Docs. http://dev.apollodata.com/react/. Accessed: 15-06-2017.

[43] Wolfram|Alpha: Materials. https://www.wolframalpha.com/examples/Materials.html. Accessed: 19-06-2017.

[44] Redux. http://redux.js.org/docs/introduction/. Accessed: 19-06-2017.

[45] Relay. https://facebook.github.io/relay/. Accessed: 19-06-2017.

[46] Mongodb for giant ideas. https://www.mongodb.com/. Accessed: 19-06-2017.

[47] Mongoose. http://mongoosejs.com/. Accessed: 19-06-2017.

[48] Git. https://git-scm.com/. Accessed: 19-06-2017.

[49] React Native: Listview. https://facebook.github.io/react-native/docs/listview.html. Accessed: 19-06-2017.

[50] React Native: Flatlist. https://facebook.github.io/react-native/docs/flatlist.html. Accessed: 19-06-2017.

[51] Navigation Drawer - Patterns. https://material.io/guidelines/patterns/navigation-drawer.html. Accessed: 19-06-2017.

[52] Apple Inc. Action Sheets - Views - iOS Human Interface Guidelines.
https://developer.apple.com/ios/human-interface-guidelines/ui-views/action-sheets/.
Accessed: 19-06-2017.

[53] React Native (v0.41): Navigation.
http://facebook.github.io/react-native/releases/0.41/docs/navigation.html.
Accessed: 19-06-2017.

[54] React Native (v0.44): Navigation 0.44.
http://facebook.github.io/react-native/releases/0.44/docs/navigation.html.
Accessed: 19-06-2017.

[55] React Navigation.
https://reactnavigation.org/.
Accessed: 19-06-2017.

[56] React Native: AsyncStorage.
https://facebook.github.io/react-native/docs/asyncstorage.html. Accessed: 19-06-2017.

[57] React Native: Animated.
https://facebook.github.io/react-native/docs/animated.html. Accessed: 19-06-2017.

[58] Gmail App – Android Google Play.
https://play.google.com/store/apps/details?id=com.google.android.gm.
Accessed: 19-06-2017.

[59] Spencer Ahrens. Better List Views in React Native.
https://facebook.github.io/react-native/blog/2017/03/13/better-list-views.html, March 2017.
Accessed: 23-06-2017.

[60] Analyze Runtime Performance | Google Developers.
https://developers.google.com/web/tools/chrome-devtools/rendering-tools/.
Accessed: 19-06-2017.

[61] Shoutem UI (GitHub).
https://github.com/shoutem/ui. Accessed: 19-06-2017.

[62] Native Elements (GitHub).
https://github.com/react-native-training/react-native-elements. Accessed: 19-06-2017.

[63] Native Base (GitHub).
https://github.com/GeekyAnts/NativeBase. Accessed: 19-06-2017.

[64] React Native: StyleSheet.
https://facebook.github.io/react-native/docs/stylesheet.html. Accessed: 19-06-2017.

[65] Color - iOS Human Interface Guidelines.
https://developer.apple.com/ios/human-interface-guidelines/visual-design/color.
Accessed: 19-06-2017.

[66] Color - Material Design Guidelines.
https://material.io/guidelines/style/color.html. Accessed: 19-06-2017.

[67] React Native Vector icons.
https://github.com/oblador/react-native-vector-icons.
Accessed: 12-06-2017.

[68] React Native Charts Wrapper.
https://github.com/wuxudong/react-native-charts-wrapper.
Accessed: 11-06-2017.

[69] React Native: DrawerLayout.
https://facebook.github.io/react-native/docs/drawerlayoutandroid.html.
Accessed: 19-06-2017.

[70] Visual Studio Code - Code Editing. Redefined.
https://code.visualstudio.com/. Accessed: 19-06-2017.

[71] React Native Tools - Visual Studio Marketplace.
https://marketplace.visualstudio.com/items?itemName=vsmobile.vscode-react-native.
Accessed: 12-06-2017.

[72] ngrok - Secure tunnels to localhost.
https://ngrok.com/. Accessed: 14-06-2017.

[73] GitHub - Build software better, together.
https://github.com/. Accessed: 13-06-2017.

[74] Josh Owens. State Of Javascript Survey Results: Mobile Frameworks.
http://stateofjs.com/2016/mobile/. Accessed: 15-06-2017.

[75] React Native: ActivityIndicator.
https://facebook.github.io/react-native/docs/activityindicator.html. Accessed: 19-06-2017.